8.8 NONBLOCKING POINT-TO-POINT COMMUNICATION

The point-to-point communication calls introduced in Sections 8.5.1 and 8.5.3, MPI_Send and MPI_Recv, do not return from the respective function call until the send and receive operations have completed. While this ensures that the send and receive buffers used in the MPI_Send and MPI_Recv arguments are safe to use or reuse after the function call, it also means that unless there is a simultaneously matching send for each receive, the code will deadlock, resulting in the code hanging. This common type of bug is examined in Chapter 14. One way to avoid this is by using nonblocking point-to-point communication.

Nonblocking point-to-point communication returns immediately from the function call before confirming that the send or the receive has completed. These nonblocking calls are MPI_Isend and MPI_Irecv. They are used coupled with MPI_Wait, which will wait until the operation is completed. When querying whether a nonblocking point-to-point communication has completed, MPI_Test is often paired with MPI_Isend and MPI_Irecv. Nonblocking point-to-point calls can simplify code development to avoid such deadlocks more easily and also potentially enable the overlap of useful computation while checking to see if the communication has completed.

The syntax of each of these calls is the same as for the blocking calls except for the addition of a request argument and the elimination of the status output in the MPI_Recv arguments.

```
int MPI_Isend (void *message, int count, MPI_Datatype datatype, int dest, int tag,
 MPI_Comm comm, MPI_Request *send_request)
```

```
int MPI_Irecv (void *message, int count, MPI_Datatype datatype, int source, int tag,
 MPI_Comm comm, MPI_Request *recv_request)
```

Because both MPI_Isend and MPI_Irecv return immediately after calling without confirming that the message-passing operations have completed, the application user needs a way to specify when these operations must complete. This is done with MPI_Wait:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

When MPI_Wait is called, the nonblocking request originating from MPI_Isend or MPI_Irecv is provided as an argument. The status that was previously provided directly from MPI_Recv is now supplied as an output from MPI_Wait.

Similar to MPI_Wait, MPI_Test can be paired with an MPI_Isend or MPI_Irecv call to query whether the message passing has completed while performing other work. MPI_Test shares similar syntax to MPI_Wait, adding only a flag that is set to true if the request being queried has completed.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 301

An example of using nonblocking communication is presented in Code 8.12. In this example, the send commands are issued first, followed by the receive commands. If using blocking communication and sending a sufficiently large message this would normally result in a deadlock, but nonblocking communication avoids this pitfall.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <mpi.h>
4
5 int main(int argc, char* argv[]) {
6  int a, b;
7  int size, rank;
8  int tag = 0; // Pick a tag arbitrarily
9  MPI_Status status;
10 MPI_Request send_request, recv_request;
11
12 MPI_Init(&argc, &argv);
13 MPI_Comm_size(MPI_COMM_WORLD, &size);
14 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16 if (size != 2) {
17   printf("Example is designed for 2 processes\n");
18   MPI_Finalize();
19   exit(0);
20 }
21 if (rank == 0) {
22   a = 314159; // Value picked arbitrarily
23
24   MPI_Isend(&a, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &send_request);
25   MPI_Irecv (&b, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &recv_request);
26
27   MPI_Wait(&send_request, &status);
28   MPI_Wait(&recv_request, &status);
29   printf ("Process %d received value %d\n", rank, b);
30
31 } else {
32
33   a = 667;
34
35   MPI_Isend (&a, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &send_request);
36   MPI_Irecv (&b, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &recv_request);
37
38   MPI_Wait(&send_request, &status);
39   MPI_Wait(&recv_request, &status);
40   printf ("Process %d received value %d\n", rank, b);
41 }
42
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 302

```
43 MPI_Finalize();
44 return 0;
45 }
```

Code 8.12. Example of nonblocking point-to-point communication. Process 0 sends the integer 314159 to process 1 while process 1 sends the integer 667 to process 0. The particular order of the listing of Isend and Irecv in lines 24–25 and 35–36 does not matter because the calls are nonblocking.

```
> mpirun -np 2 ./code12
Process 0 received value 667
Process 1 received value 314159
```

8.9 USER-DEFINED DATA TYPES

Application developers will frequently wish to create a user-defined data type built out of the predefined MPI types listed in Table 8.1. This is accomplished using MPI_Type_create_struct and MPI_Type_commit.

```
MPI_Type_create_struct(int number_items,
                       const int *blocklengths,
                       const MPI_Aint *array_of_offsets,
                       const MPI_Datatype *array_of_types,
                       MPI_Datatype *new_datatype_name)
```

```
MPI_Type_commit(MPI_Datatype *new_datatype_name)
```

Creating a user-defined data type consists of providing the number of different partitions of existing MPI data-type elements (*number_items*), three separate arrays of length *number_items* containing the number of elements per block, byte offsets of each block and the MPI data types of each block, and the new name for the user-defined type. This name is then passed as an argument to MPI_Type_commit, after which it can be used in all existing MPI functions.

An example of creating a user-defined data type from a C struct and broadcasting it to all processes is provided in Code 8.13. In this example, a C struct containing some typical variable names for a simulation is populated with values on process 0. The user-defined data type for this C struct, *mpi_par*, is created and committed on lines 38–39. The values for the structure are then broadcast to all other processes in line 41.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 303

```
1 #include <stdio.h>
2 #include <stddef.h>
3 #include "mpi.h"
4
5 typedef struct {
6  int max_iter;
7  double t0;
8  double tf;
9  double xmin;
10 } Pars;
11
12 int main(int argc,char **argv) {
13
14  MPI_Init(&argc,&argv);
15  int rank;
16  int root = 0; // define the root process
17  MPI_Comm_rank(MPI_COMM_WORLD,&rank); // identify the rank
18
19  Pars pars;
20  if ( rank == root ) {
21    pars.max_iter = 10;
22    pars.t0 = 0.0;
23    pars.tf = 1.0;
24    pars.xmin = -5.0;
25  }
26
27  int nitems = 4;
28  MPI_Datatype types[nitems];
29  MPI_Datatype mpi_par; // give my new type a name
30  MPI_Aint offsets[nitems]; // an array for storing the element offsets
31  int blocklengths[nitems];
32
33  types[0] = MPI_INT; offsets[0] = offsetof(Pars,max_iter);blocklengths[0] = 1;
34  types[1] = MPI_DOUBLE; offsets[1] = offsetof(Pars,t0);blocklengths[1] = 1;
35  types[2] = MPI_DOUBLE; offsets[2] = offsetof(Pars,tf);blocklengths[2] = 1;
36  types[3] = MPI_DOUBLE; offsets[3] = offsetof(Pars,xmin);blocklengths[3] = 1;
37
38  MPI_Type_create_struct(nitems,blocklengths,offsets,types,&mpi_par);
39  MPI_Type_commit(&mpi_par);
40
41  MPI_Bcast(&pars,1,mpi_par,root,MPI_COMM_WORLD);
42
43  printf("Hello from rank %d; my max_iter value is %d\n",rank,pars.max_iter);
44
45  MPI_Finalize();
46
47  return 0;
48 }
```

Code 8.13. Example of creating and using a user-defined data type in an MPI collective.

```
> mpirun -np 4 ./code13
Hello from rank 0: my max_iter value is 10
Hello from rank 2: my max_iter value is 10
Hello from rank 1: my max_iter value is 10
Hello from rank 3: my max_iter value is 10
```

8.10 SUMMARY AND OUTCOMES OF CHAPTER 8

- There was probably no greater achievement of practical utility for the advancement of HPC than the development of MPI.
- MPI is a community-driven specification that continues to evolve.
- MPI is a library with an API, not a language.
- MPICH was the first reduction to practice the MPI standard.
- Key elements of MPI are point-to-point communication and collective communication.
- MPI has a set of predefined data types for use in library calls.
- Point-to-point communication calls are typified by the MPI_Send and MPI_Recv calls.
- Collective communication is typified by the broadcast, gather, and scatter operations.
- Important extensions of these collective operations are allgather, reduce, and alltoall.
- Nonblocking point-to-point communications are frequently used to simplify code development and avoid deadlocks.
- User-defined data types can be built up starting from existing MPI data types and used in MPI function calls.

8.11 EXERCISES

1. Modify Code 8.13 to send and receive the user-defined data type *mpi_par* multiple times between two processes. Add an integer to the Par struct to count how many times the data have been passed back and forth.
2. Modify Code 8.13 so that each process sends *mpi_par* to the process with rank + 2 and receives data from rank − 2. For example, if there were 16 processes, process 0 would send to process 2 and receive from process 14 while process 1 would send to process 3 and receive from process 15.
3. Write a distributed matrix−vector multiplication code using MPI. Use a dense matrix and a dense vector. Call C language Basic Linear Algebra Subprograms (CBLAS) on each process for the local matrix−vector multiplication.
4. Rewrite Code 8.11 using point-to-point communication. Generalize the code to run on an arbitrary number of processes. Compare the performance of MPI_Alltoall with your point-to-point communication implementation.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 305

5. Rewrite Code 8.9 so that the global vector sizes stay the same as the number of processes varies. This will require changing the local vector size depending on the number of processes on which MPI is launched. Plot the time to solution for your code as a function of the number of processes for various global vector sizes.

REFERENCES

[1] MPI Forum, MPI Standardization Forum, [Online]. http://mpi-forum.org/.
[2] Argonne National Laboratory, MPICH, [Online]. www.mpich.org.
[3] B. Kernighan, D. Ritchie, The C Programming Language, Prentice Hall, s.l., 1988.

CHAPTER

9

PARALLEL ALGORITHMS

CHAPTER OUTLINE

CHAPTER OUTLINE 285

- 9.1 Introduction 286
- 9.2 Fork–Join 287
- 9.3 Divide and Conquer 291
- 9.4 Manager–Worker 292
- 9.5 Embarrassingly Parallel 294
- 9.6 Halo Exchange 295
  - 9.6.1 The Advection Equation Using Finite Difference 295
  - 9.6.2 Sparse Matrix Vector Multiplication 297
- 9.7 Permutation: Cannon's Algorithm 301
- 9.8 Task Dataflow: Breadth First Search 306
- 9.9 Summary and Outcomes of Chapter 9 310
- 9.10 Exercises 311
- References 311

9.1 INTRODUCTION

Modern supercomputers employ several different modalities of operation to take advantage of parallelism both to exploit enabling technologies and to contribute to achieving the highest performance possible across a wide spectrum of algorithms and applications. Three of these most common hardware architecture forms present in a supercomputer are single-instruction multiple data (SIMD) parallelism, shared memory parallelism, and distributed memory parallelism. Shared memory and distributed memory parallelism are subclasses of the multiple-instruction multiple data (MIMD) class of Flynn's computer architecture taxonomy.

Each of these modalities is present in a modern supercomputer. While the unifying theme of a parallel computer architecture is parallelism, the ways in which a parallel algorithm exploits physical parallelism in each of these modalities can differ substantially. Some parallel algorithms are better suited for one kind of parallelism versus another. Often a completely different parallel algorithm will be needed, depending on the targeted parallel computer architecture structure. Frequently a combination of all three will be necessary for a parallel algorithm to achiee the highest possible performance that a supercomputer can provide.

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00009-5

Copyright © 2018 Elsevier Inc. All rights reserved.

Table 9.1 Examples of Generic Classes of Parallel Algorithms

Generic Class of Parallel Algorithm	Example
Fork–join	OpenMP parallel for-loop
Divide and conquer	Fast Fourier Transform, parallel sort
Halo exchange	Finite difference/finite element partial differential equation solvers
Permutation	Cannon's algorithm, Fast Fourier Transform
Embarrassingly parallel	Monte Carlo
Manager–worker	Simple adaptive mesh refinement
Task dataflow	Breadth first search

In 2004 an influential set of seven classes of numerical methods commonly used on supercomputers were identified [1]. These are known as the "seven dwarfs" or "seven motifs": dense linear algebra, sparse linear algebra, spectral methods, N-body methods, structured grids, unstructured grids, and Monte Carlo methods. These seven classes of algorithms represent a large segment of supercomputing applications today and many high performance computing (HPC) benchmarks are built specifically to target them. In addition to the original "seven dwarfs", researchers have added other important emerging classes of numerical methods found in supercomputing applications, including graph traversal, finite state machines, combinational logic, and statistical machine learning [2]. Optimally mapping these numerical methods to a parallel algorithm implementation is a key challenge for supercomputing application developers.

Several classes of parallel algorithms share key characteristics and are driven by the same underlying mechanism from which the parallelism is derived. Some examples of these generic classes of parallel algorithms include fork–join, divide and conquer, manager–worker, embarrassingly parallel, task dataflow, permutation, and halo exchange. Some examples of each class are listed in Table 9.1.

This chapter examines a wide variety of parallel algorithms and the means by which the parallelism is exposed and exploited. While the specific implementation of the algorithm for SIMD or MIMD parallel computer architectures will differ, the conceptual basis for extracting parallelism from the algorithm will not. The chapter begins by examining fork–join type parallel algorithms and an example from the divide-and-conquer class of parallel algorithms, parallel sort. Examples from manager–worker type algorithms and a specific subclass of it, embarrassingly parallel, are then examined. Halo-exchange parallel algorithm examples are examined also including the advection equation and sparse matrix vector multiplication. A permutation example of Cannon's algorithm and a task dataflow example of a breadth first search algorithm complete the chapter.

9.2 FORK–JOIN

The fork–join parallel design pattern is a key component of the OpenMP execution model presented in Chapter 7 and is frequently employed in programming models targeting shared memory parallelism. In

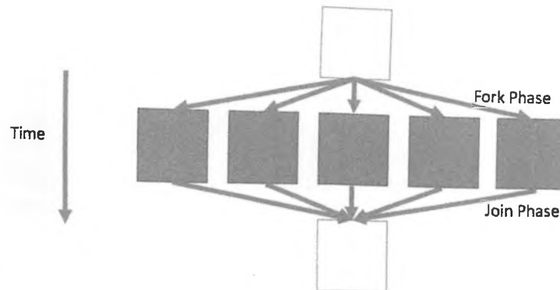


FIGURE 9.1

This is an illustration of the fork–join parallel design pattern frequently used in parallel algorithms intended for shared memory parallelism. The *empty boxes* indicate work that is serial (i.e., not parallelizable), while the *filled boxes* indicate work that can be performed concurrently. In the "fork" phase, concurrent operators known as threads (denoted here by the branching *lines*) are created to perform the concurrent work. In the "join" phase, the results of those concurrent operators are accumulated into a single resulting operator.

regions of a sequential algorithm where work can proceed concurrently, a group of lightweight concurrent operators, frequently called "threads", are created to perform that work. Once the work is completed, the results from each of these operators are accumulated during the "join" phase. This process is illustrated in Fig. 9.1.

The OpenMP parallel for-loop construct is a simple example of this type of parallel algorithm. Consider the example of parallel work sharing presented in Code 1. A previously initialized array b is added to another expression to initialize array a. Because each work element in the for-loop (see line 3) is independent of every other element, the work in this loop can proceed concurrently. Consequently, a parallel for-loop construct is added in line 1. Fig. 9.2 illustrates the fork–join behavior of the resulting concurrent operators.

```
1 #pragma omp parallel for
2   for (i=0;i<30;i++)
3     a[i] = b[i] + sin(i)
```

Code 1: An OpenMP example of fork–join for work sharing.

While the fork–join parallel design pattern is the main execution model for OpenMP, it is also found in other parallel programming models, especially those which target shared memory parallelism.

9.3 DIVIDE AND CONQUER

Algorithms denoted as "divide and conquer" break a problem into smaller subproblems which share similar enough algorithmic properties to the original problem that they can in turn also be subdivided. Using recursion, the larger problem is broken down into small enough pieces that it can

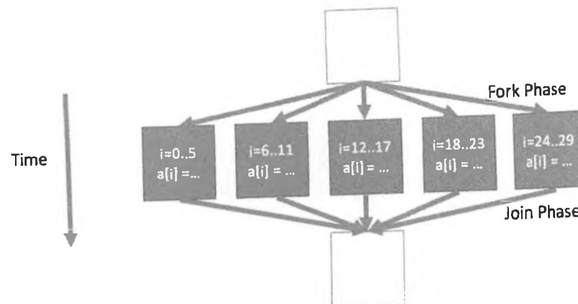


FIGURE 9.2

The resulting fork–join structure from the work-sharing example in Code 9.1 for five threads. Each concurrent operator performs an independent computation that is then joined into the final serial operator.

be easily solved with minimal computation. Because the original problem has been broken down into several smaller computations that are independent of one another, there is a natural concurrency for exploiting parallel computation resources. Frequently, divide-and-conquer type algorithms are also naturally parallel algorithms because of this concurrency and, like fork–join type algorithms, can perform very well on shared memory architectures. On distributed memory architectures, however, network latency and load imbalance can complicate the direct application of divide-and-conquer type algorithms.

One well-studied example of a divide-and-conquer algorithm with natural concurrency is quicksort [3]. As a sorting algorithm, it aims to sort a list of numbers in order of increasing value. To start, a random element of the array is selected to serve as a pivot point. Using this pivot, the rest of the list is divided into a list containing numbers smaller than the pivot and a list containing numbers larger than the pivot. This process is then repeated recursively for each of the two lists. Upon completion of recursion the resulting sorted child subproblems are concatenated for the final result. An example is given in Fig. 9.3.

The efficiency of the algorithm is significantly impacted by which element is chosen as the pivot point. If the array has N data items, the worst-case performance will be proportional to N^2; however, for most cases the performance is much faster, proportional to $N \log N$. Because the two branched lists in quicksort can be sorted independently, there is a natural concurrency of computation that can be used for parallelization. On a distributed memory architecture, exploiting this concurrency incurs a significant communication cost as sorted lists are passed from one process to another during recursion. This makes direct application of quicksort on a distributed memory architecture undesirable. However, a modification to the approach based on sampling can be made to improve this situation.

The regular sampling parallel sort algorithm is designed for better performance on distributed memory architectures with quicksort underlying the approach [4]. The algorithm is detailed below.

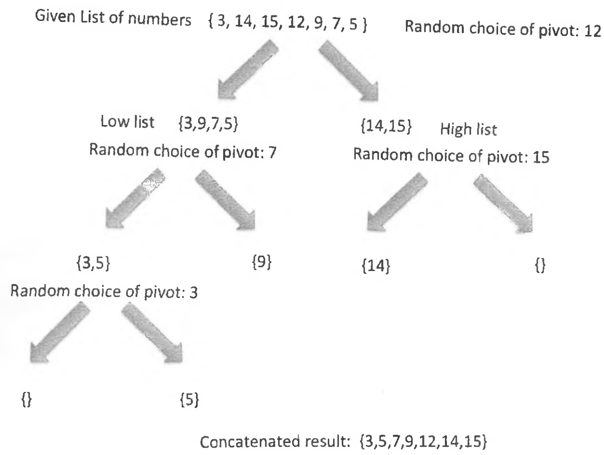


FIGURE 9.3

Example of serial quicksort algorithm.

- An array of numbers to be sorted is distributed equally among P processes. Thus if the array size is N, each process will have N/P local elements.

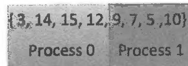


FIGURE 9.4A

- Each process runs sequential quicksort on its local data.

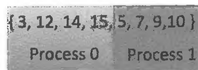


FIGURE 9.4B

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 311

- The resulting sorted arrays are sampled at intervals determined by the global array size N and the number of processes P. Samples are taken at every N/P^2 location starting at 0, i.e.,: array element indices 0, N/P^2, $2N/P^2$, ..., $(P - 1)\,N/P^2$ form the sample array from each local data.

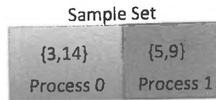


FIGURE 9.4C

- The resulting samples are gathered to a root process and sorted sequentially with quicksort.

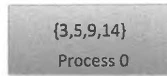


FIGURE 9.4D

- Regularly sampled $P - 1$ pivot values computed from the sample set are broadcast to the other processes. Thus N/P^2, $2N/P^2$, ..., $(P - 1)\,N/P^2$ indices form the sample $P - 1$ pivot points. In this example, the only pivot point broadcast is 9.

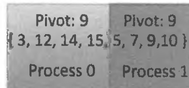


FIGURE 9.4E

- Each process divides its sorted segment of the array into P segments using the broadcast $P - 1$ pivot values.

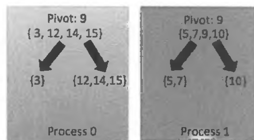


FIGURE 9.4F

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 312

- Each process performs an all-to-all operation on the P segments. Thus the ith process keeps the ith segment and sends the jth segment to the jth process.

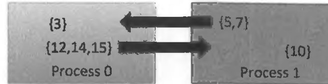
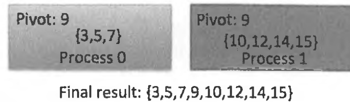


FIGURE 9.4G

- The arriving segments are merged into a single list and locally sorted.



Final result: {3,5,7,9,10,12,14,15}

FIGURE 9.4H

An example of this algorithm for $P = 2$ and $N = 8$ is illustrated in Fig. 9.4A–H.

9.4 MANAGER–WORKER

Manager–worker incorporates two different workflows in its execution: one intended for execution by just one process called the manager process, and another intended for execution by several other processes called worker processes. This approach has also historically been called "master–slave". Applications that are dynamic in nature frequently use this type of parallel design algorithm so that the manager process can coordinate and issue task actions to worker processes in response to changes in a simulation outcome. Many adaptive mesh refinement applications also use this parallel design algorithm because the meshes and data placement patterns change in response to a solution value. Such an adaptive mesh refinement is illustrated in Fig. 9.5. Manager–worker codes frequently take the form illustrated in Code 9.2, where an "if" statement distinguishes the workflow between manager and worker tasks.

```
1 if ( my_rank == master ) {
2    send_action(INITIALIZE);
3
4    for (int i=0;i<num_timesteps;i++) {
5      send_action(REFINE);
6      send_action(INTEGRATE);
7      send_action(OUTPUT);
8    }
9 } else {
10   listen_for_actions();
11 }
```

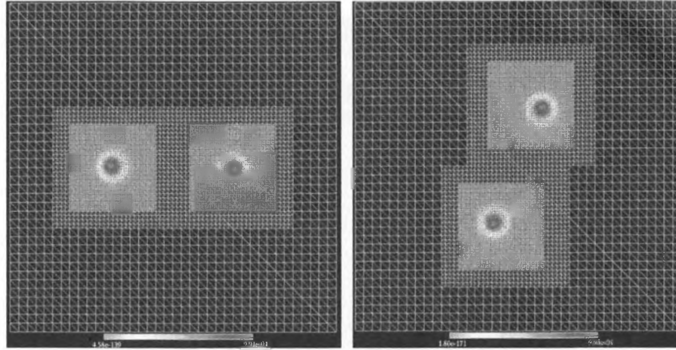


FIGURE 9.5

Example of a manager—worker adaptive mesh refinement code evolving a dynamic system of two compact objects orbiting each other. The meshes follow the orbiting compact objects as they move clockwise in the computational domain.

Code 9.2 is manager—worker example code adapted from the adaptive mesh refinement code used to generate Fig. 9.5. The manager process (called "master" in this example) directs the refinement characteristics and sends actions to worker processes that are always listening for additional instructions from the manager.

9.5 EMBARRASSINGLY PARALLEL

The term "embarrassingly parallel" is a common phrase in scientific computing that is both widely used and poorly defined. It suggests lots of parallelism with essentially no intertask communication or coordination, as well as a highly partitionable workload with minimal overhead. In general, embarrassingly parallel algorithms are a subclass of manager—worker algorithms. They are called embarrassingly parallel because the available concurrency is trivially extracted from the workflow. These algorithms sometimes require reduction operation at the end to gather the results into a manager process. While this does require some minimal coordination and intertask communication, these "almost embarrassingly parallel" algorithms are still generally referred to as embarrassingly parallel.

Monte Carlo simulations mainly fall into the category of embarrassingly parallel. Monte Carlo methods are statistical approaches for studying systems with a large number of coupled degrees of freedom, modeling phenomena with significant uncertainty in the inputs, and solving partial differential equations with more than four dimensions. Computing the value of π is a simple example.

- Define a square domain and inscribe a circle inside that domain.
- Randomly generate the coordinates of points lying inside the square domain; count the points that also lie in the circle.
- $\pi/4$ is the ratio of the number of points that lie in the circle to the total number of random points generated.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 314

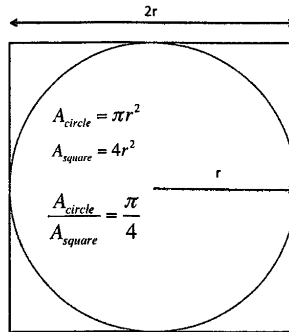


FIGURE 9.6

When generating random coordinates inside a square, the ratio of the number of points lying inside an inscribed circle to the total number of random points will be $\pi/4$.

The reasoning behind this algorithm is as follows. A circle with radius r inscribed in a square will have an area of πr^2 while the square will have an area of $(2r)^2 = 4r^2$, as seen in Fig. 9.6. The ratio of the area of the circle to the area of the square will also be the probability that a random point generated in the square lies in the circle. The ratio of these two areas is $\pi/4$.

The parallel version of this algorithm is illustrated in Fig. 9.7.

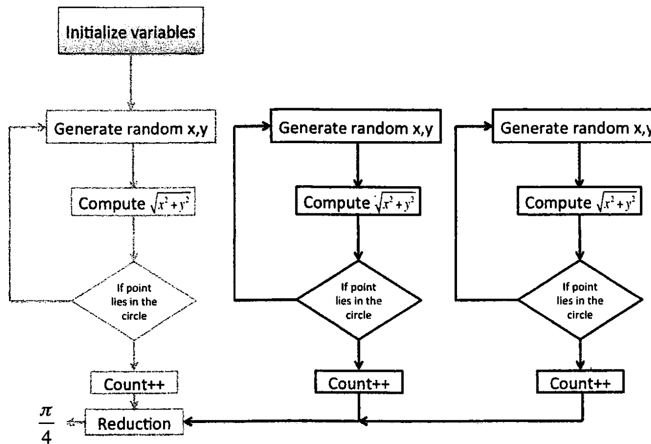


FIGURE 9.7

Embarrassingly parallel example: computing π using statistical methods. The manager is in light gray while the various workers are in black.

9.6 HALO EXCHANGE

Many parallel algorithms fall into a problem class where every parallel task is executing the same algorithm on different data without any manager algorithm present. This is sometimes referred to as the data parallel model. Data parallelism is frequently used in applications that are static in nature because a computational task can be mapped to particular subset of data throughout the life of the simulation. However, in all but the most simple of applications, some information in each data subset mapped to the parallel task has to be exchanged and synchronized for the application algorithm to function properly. This exchange of intertask information is called halo exchange.

As the name implies, a halo is a region exterior to the data subset mapped to a parallel task. It acts as an artificial boundary to that data subset and contains information that originates from the data subsets of neighboring parallel tasks. A halo is illustrated in Fig. 9.8.

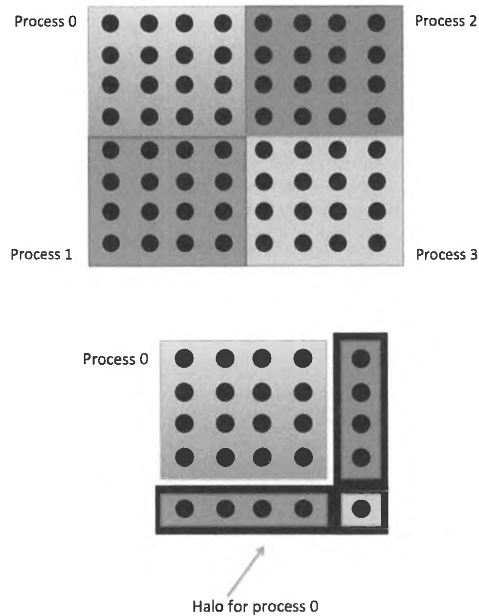


FIGURE 9.8

Illustration of a one-deep halo. In this illustration, various data points (colored dark) are split across four different processes (top figure). For each process there are two boundaries in the data that are interprocess boundaries. For process 0, these are the right and bottom edges of the square. A one-deep halo for process 0 (bottom figure) consists of those data points that are closest to the interprocess boundary of process 0 but not mapped to process 0.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 316

Halo exchange enables each task to perform computations and update the subset of data mapped to that task while having access to any data necessary for such computations that may not be local. Halo exchange is extremely common in parallel toolkits for solving partial differential equations and in linear algebra computations. Two parallel algorithm examples are presented in this section using halo exchange: the advection equation and sparse matrix vector multiplication.

9.6.1 THE ADVECTION EQUATION USING FINITE DIFFERENCE

Wavelike phenomena permeate nature: examples include light, sound, gravitation, fluid flow, and weather, to name just a few. The study of wavelike phenomena is ubiquitous in supercomputing systems and is frequently modeled using a *partial differential equation*, or an expression involving derivatives taken against different independent variables. One of the simplest ways to solve these wavelike partial differential equations on a supercomputer is through the use of finite differencing and halo exchange. Finite differencing involves replacing the derivative expressions in the partial differential equation with approximations originating from estimating the slope between neighboring points on a uniform grid.

As an example of this parallel algorithm, consider the advection equation in Eq. (9.1).

$$\frac{\partial f}{\partial t} = -v\frac{\partial f}{\partial x} \tag{9.1}$$

This advection equation transports a scalar field $f(x, t)$ toward increasing x with speed v. The analytic solution to this partial differential equation is

$$f(x,t) = F(x - vt) \tag{9.2}$$

where $F(x)$ is an arbitrary function describing the initial condition of the system. So if the initial condition of the wavelike phenomenon for solution is

$$F(x) = e^{-x^2} \tag{9.3}$$

then the analytic solution to Eq. (9.1) would be

$$f(x,t) = e^{-(x-vt)^2} \tag{9.4}$$

as plotted in Fig. 9.9.

While the advection equation in Eq. (9.1) can be solved analytically with the solution shown in Fig. 9.9, a parallel algorithm based on halo exchange can be crafted to solve this equation numerically. The left- and right-hand partial derivatives in Eq. (9.1) are replaced with finite difference approximations to those derivatives:

$$\frac{f_i^{n+1} - f_i^n}{dt} = -v\frac{f_{i+1}^n - f_i^n}{dx} \tag{9.5}$$

where the field $f(x,t)$ has been discretized to a uniform mesh in which the x points are separated by distance dx and the time points are separated by time dt, with the subscript to f indicating the spatial location in that mesh and the superscript to f indicating the temporal location in that mesh. This is illustrated in Fig. 9.10.

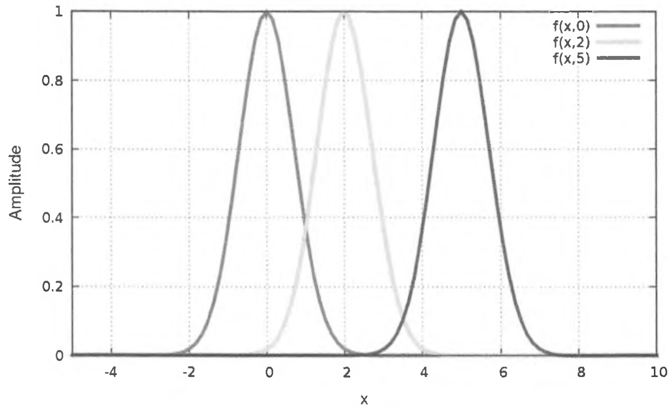


FIGURE 9.9

The solution to the advection equation given in Eq. (9.1) with the initial condition in Eq. (9.3) and velocity set to be 1. The solution at several times ($t = 0$, 2, and 5) is plotted. The scalar field is transported to the right as time increases.

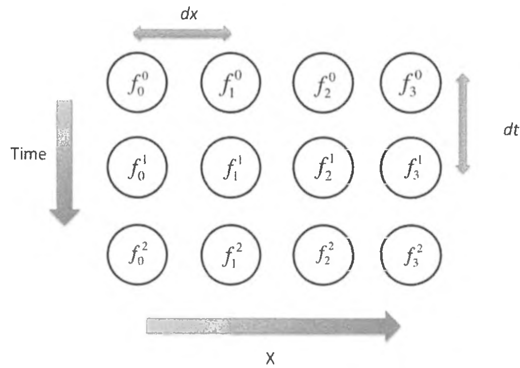


FIGURE 9.10

The scalar field values are assigned to discrete mesh points in time and space, with points separated from one another by value dx in the x direction and dt in the time direction. The superscript indicates the time index (0–2 in this illustration) and the subscript indicates the spatial index in the x direction (0–3 in this illustration).

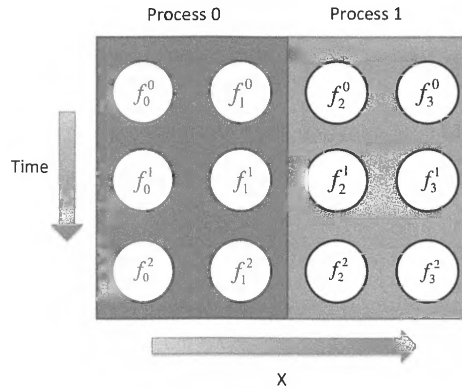


FIGURE 9.11

The discretized mesh is partitioned across several processes. As an example of a data parallelism, the same computation from the right-hand side of Eq. (9.6) is concurrently applied to different data in process 0 and 1 starting at the first row of values corresponding to the initial time.

Because the initial condition of the scalar field, f_i^0 or $f(x,0)$, is known, algebraic manipulation of Eq. (9.5) enables all future time values to be found iteratively using Eq. (9.6) provided that the right-hand side of the expression can be computed.

$$f_i^{n+1} = f_i^n - v\frac{dt}{dx}\left(\frac{f_{i+1}^n - f_i^n}{dx}\right) \tag{9.6}$$

To compute the right-hand side of Eq. (9.6) in parallel, the discretized mesh is partitioned across several processes, as illustrated in Fig. 9.11. This is an example of data parallelism where the same operation, computing the right-hand side of Eq. (9.6) in this case, is applied to different data spread across several processes.

To compute the right-hand side for the data in process 0, however, some information is needed from process 1. This information is provided through halo exchange, as illustrated in Fig. 9.12.

The parallel algorithm is summarized in Fig. 9.13.

9.6.2 SPARSE MATRIX VECTOR MULTIPLICATION

Parallel algorithms designed around halo exchange frequently show up not just in mesh-based solvers, as seen in Section 9.6.1, but also in sparse linear algebra operations such as the sparse matrix vector multiplication used in the high performance conjugate gradients (HPCG) benchmark presented in Chapter 4.

For a matrix of size $N \times N$ and vector of size N, matrix−vector multiplication is given by Eq. (9.7):

$$x_i = \sum_{j=0}^{N-1} A_{ij} b_j \tag{9.7}$$

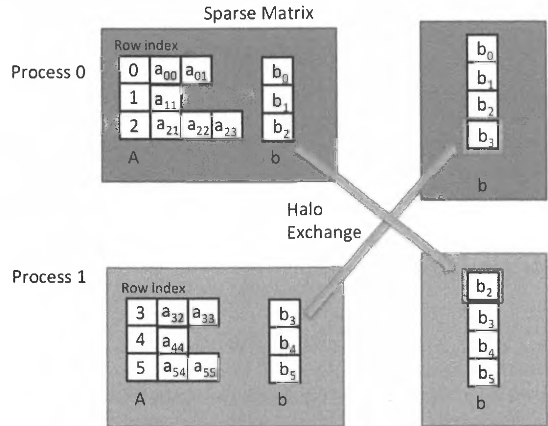


FIGURE 9.15

Illustration of the data parallel model with halo exchange for sparse matrix vector multiplication. Process 0 requires element b_3 to be available to compute Eq. (9.8), while process 1 requires element b_2. Once these vector elements are exchanged, indicated by the *arrows*, each process is able to compute Eq. (9.8) independently with local data.

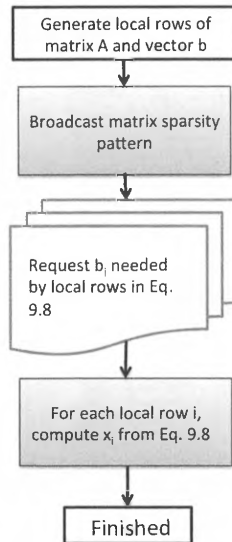


FIGURE 9.16

Summary of sparse matrix vector multiplication using a compressed sparse row format for a data parallel model with halo exchange. The halo-exchange phase is indicated in red (gray in print version).

9.7 PERMUTATION: CANNON'S ALGORITHM

Among algorithms which rely upon a data parallelism approach where the same algorithm is applied to different data to extract concurrency, a certain subclass of problem relies upon permutation routing operations to perform all-to-all operations iteratively. This type of parallel algorithm is very frequent used in applications requiring a linear algebra transpose operation or some type of matrix–matrix multiplication. In this section, one such example is explored: Cannon's algorithm for dense matrix–matrix [5].

In computational linear algebra, algorithms involving matrix operations are frequently divided into two classes: sparse and dense. Sparse matrices refer to those matrices that are dominated by zeros and generally employ some type of compression algorithm so that the zero entries are neither stored nor operated on. Dense matrices are those which are dominated by nonzero entries. Cannon's algorithm is a matrix–matrix multiplication algorithm for distributed memory parallelism designed for dense matrices, and relies heavily on permutation routing.

Matrix–matrix multiplication for two $N \times N$ matrices A and B is summarized in Eq. (9.9)

$$C_{ij} = \sum_{k=0}^{k=N-1} A_{ik} B_{kj} \tag{9.9}$$

where the subscripts indicate the row and column index of the matrix entry. To create a parallel algorithm for Eq. (9.9), a good place to start is a block algorithm that distributes subblocks of A, B, and C among processes where each subblock is of size $N/\sqrt{P} \times N/\sqrt{P}$ and P is the number of processes. This is illustrated in Fig. 9.17.

For example, computing the subblock C_{11} of the matrix–matrix product of $A \times B$ requires computing several serial matrix–matrix products each of size $N/\sqrt{P} \times N/\sqrt{P}$, as illustrated in Fig. 9.18.

For this block partitioning approach, matrix–matrix multiplication becomes a matter of orchestrating the communication and computation of the various serial subblock matrix–matrix products. This is the heart of Cannon's algorithm.

Initially the subblocks are mapped to each process, as illustrated in Fig. 9.19.

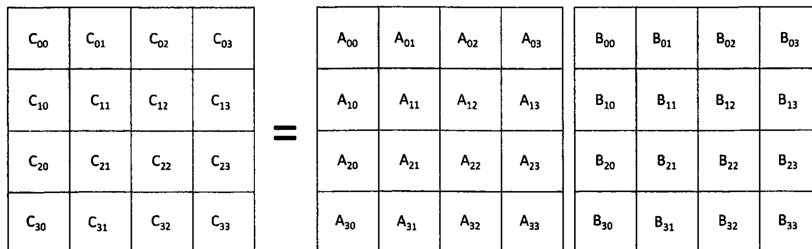


FIGURE 9.17

The global $N \times N$ matrices A and B are partitioned into P subblocks so that each subblock is of size $N/\sqrt{P} \times N/\sqrt{P}$. In this illustration, $P = 16$. Each process holds only one subblock.

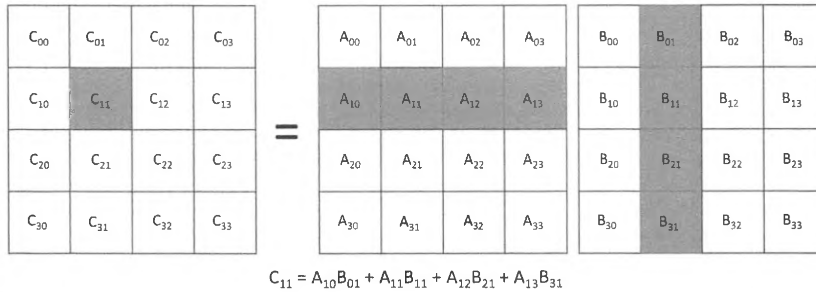


FIGURE 9.18

To compute the C_{11} subblock of the matrix–matrix product of $A \times B$, several matrix–matrix products of the highlighted subblocks must be computed. However, one block is assigned to each process, and only subblocks A_{11} and B_{11} are local to the process where C_{11} resides. All others subblocks must be communicated.

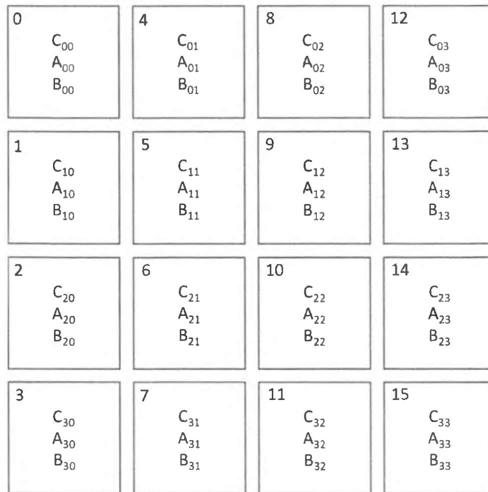


FIGURE 9.19

The subblocks are each mapped to a process for distributed memory parallelism. The process number is indicated in the upper left-hand corner in this illustration.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 322

To set up Cannon's algorithm, the A subblocks are shifted to the left while the B subblocks are shifted up, as illustrated in Figs. 9.20 and 9.21.

The memory layout after the set-up permutations is shown in Fig. 9.22.

Cannon's algorithm consists of moving matrix subblocks so that for each iteration k from 0 to 3 matrix subblocks $A_{i,(i+j+k)}$ and $B_{(i+j+k),j}$ are located on the same process as C_{ij}. For each iteration, the partial sum in Eq. (9.10) is accumulated to C_{ij}:

$$C_{ij} += A_{i,(i+j+k)} B_{(i+j+k),j} \tag{9.10}$$

where each subblock matrix–matrix multiplication uses Eq. (9.9) to compute the matrix–matrix product. The sums in Eq. (9.10), $i + j + k$, are modulus $\sqrt{P}$ (4 in this example). Thus if $(i + j + k) = 6$, the index in the matrix would become 2.

For $k = 0$, Cannon's algorithm has already been set up. For example, in Fig. 9.22 matrix C_{31} is located in the same process as matrix A_{30} and B_{01}. For every subsequent iteration of k, the A matrices have to be shifted once left and the B matrices have to be shifted up once to satisfy the condition of Eq. (9.10) and compute the partial sum. This is illustrated in Fig. 9.23.

After $\sqrt{P}$ iterations of k, the matrix–matrix product has been computed. The resulting matrices for each of the k iterations for the example are shown in Fig. 9.24. Cannon's algorithm is summarized in Fig. 9.25.

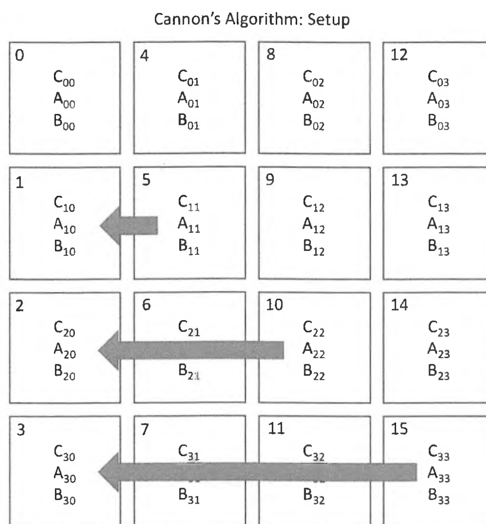


FIGURE 9.20

The A subblocks are permuted to the left to set up Cannon's algorithm.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 323

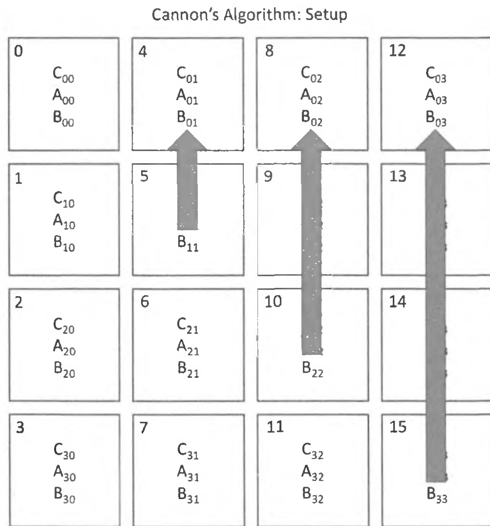


FIGURE 9.21

The B subblocks are permuted up to set up Cannon's algorithm.

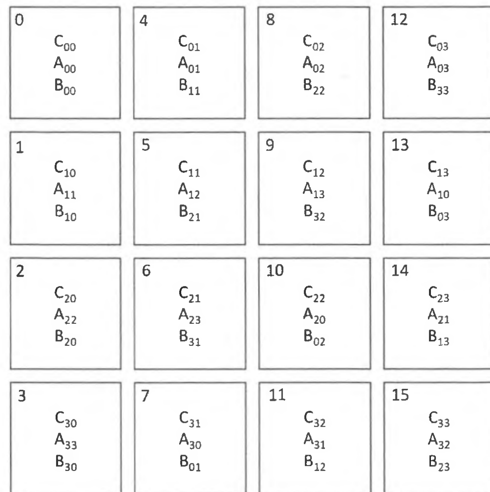


FIGURE 9.22

The layout of the matrix subblocks after performing the permutations illustrated in Figs. 9.19 and 9.20. This completes the set up of Cannon's algorithm.

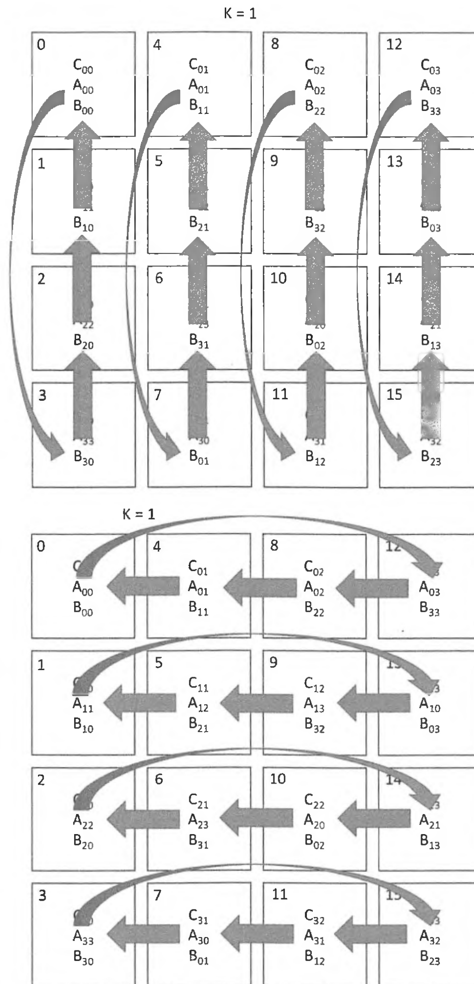


FIGURE 9.23

For each subsequent iteration of k, the B matrices are shifted up and the A matrices are shifted to the left to fulfill the condition for Eq. (9.10).

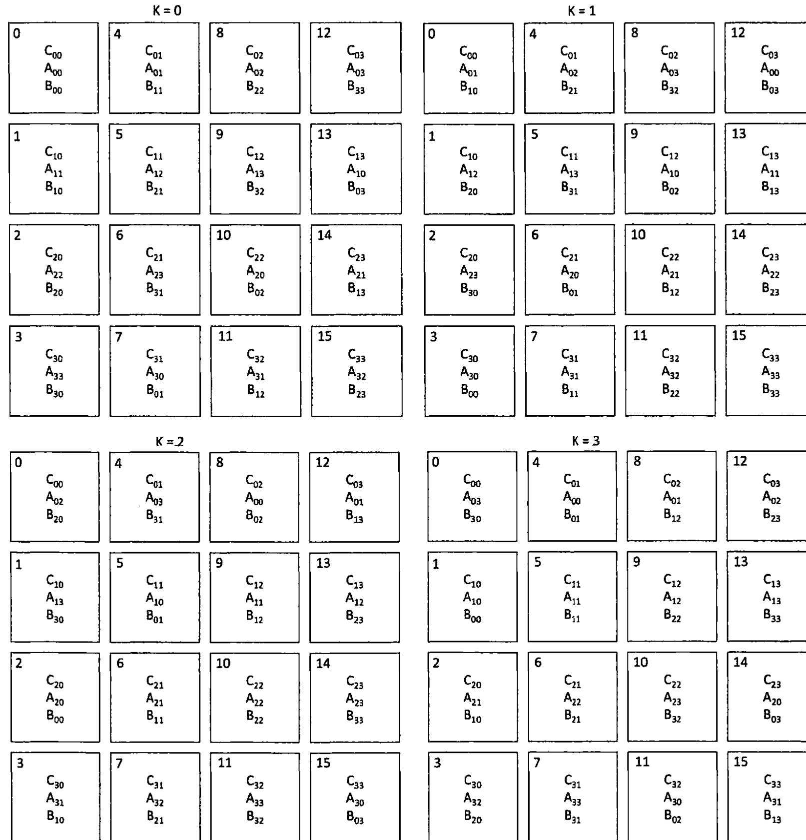


FIGURE 9.24

The distribution of the subblock matrices for each iteration of Cannon's algorithm for the example presented in Fig. 9.18.

9.8 TASK DATAFLOW: BREADTH FIRST SEARCH

The breadth first search algorithm is used for traversing graph data structures and is a key component of the Graph500 benchmark discussed in Chapter 4. A particular root vertex is given to the algorithm to start traversing the graph data structure. Each adjacent vertex to the root is then traversed and so on, thereby establishing the level (or distance) of every vertex from the root. An illustration is provided in Fig. 9.26.

While any parallel algorithm can be expressed as a graph of dependencies, many algorithms that explore graphs themselves are naturally expressed as task dataflow to maximize concurrency.

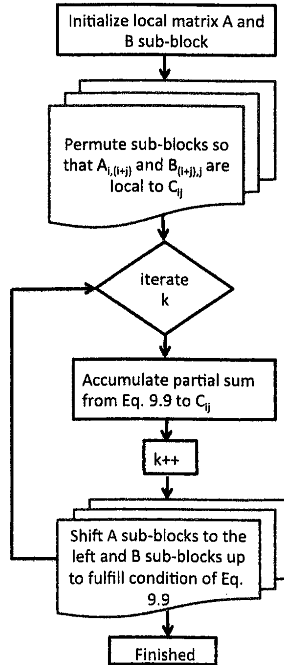


FIGURE 9.25

Summary of Cannon's algorithm for dense matrix–matrix multiplication.

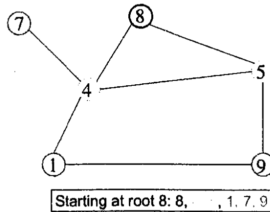


FIGURE 9.26

Example of the breadth first search traversal of this graph data structure starting at vertex 8. The adjacent vertices to the root are 4 and 5, colored light gray. The adjacent vertices to those are 1, 7, and 9, colored dark gray. *Lines* connecting the vertices are called edges.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 327

The standard parallel breadth first search algorithm is illustrated as follows.

- Each vertex list is partitioned by process with its edge list (Fig. 9.27).

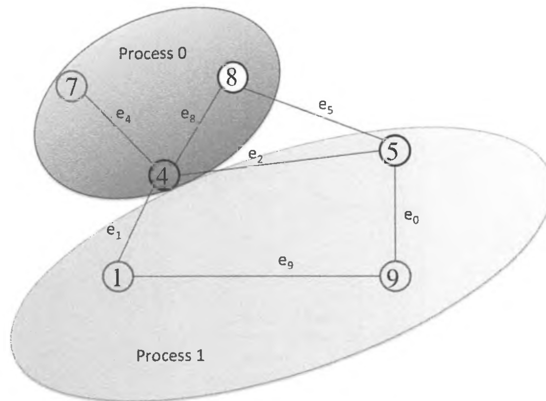


FIGURE 9.27

Partitioning the vertex list by process.

- For each vertex, associate a parent vertex and a binary flag indicating if the vertex has been visited (Fig. 9.28).

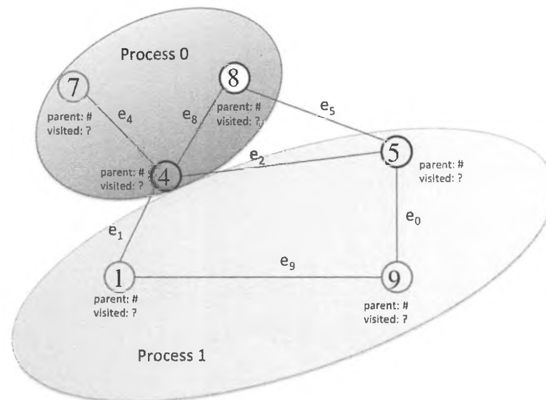


FIGURE 9.28

Adding parent vertex data and a bit to indicate if the vertex has been visited.

- In each process, scan if new vertices are visited (Fig. 9.29).

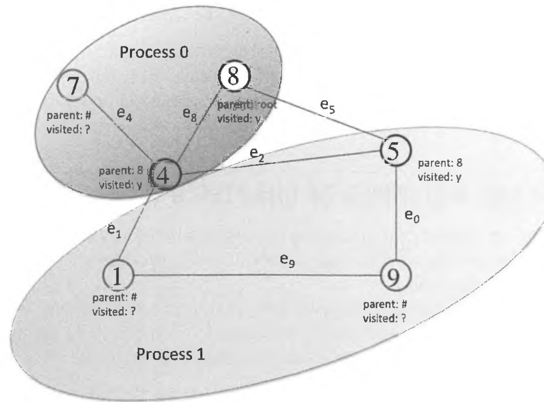


FIGURE 9.29

Each process scans if a vertex has been visited. In process 1, the scan finds that vertex 5 has been visited with parent vertex 8; in process 0, the scan finds that vertex 4 has been visited.

- For each process and each new vertex visited, follow the edge list; if the vertex is unvisited, set the parent and set to visited (Fig. 9.30).

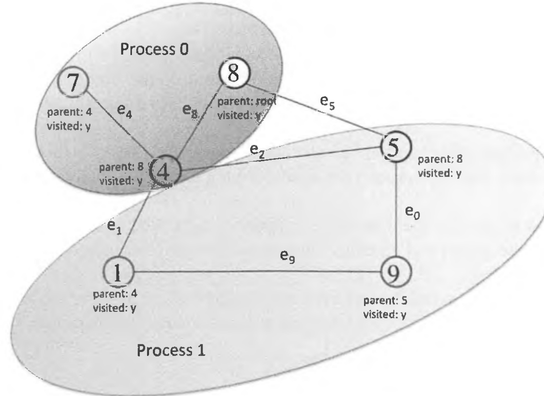


FIGURE 9.30

The edge lists for vertex 4 and vertex 5 are traversed by process 0 and process 1, respectively. Vertices 1, 7, and 9 are visited and marked accordingly. Only adjacent vertices to 4 and 5 are traversed to ensure level-wise iteration.

- Level-wise iteration is enforced with two global barriers per level, thereby ensuring no out-of-order traversals occur between processes.
- Perform an all-reduce operation at the checks to see if the algorithm has finished.

The concurrency of this breadth first search parallel algorithm is naturally tied to the edge list and the traversal tasks that result from traversing these edges. While all parallel algorithms presented in this chapter could be recast as task dataflow parallelism, many graph and knowledge management applications tend to be naturally expressed using this parallel model.

9.9 SUMMARY AND OUTCOMES OF CHAPTER 9

- Parallel algorithms are methods for organizing the computational work of a given application such that multiple parts of the workload can be performed concurrently to reduce the time to solution and increase performance.
- Fork–join parallelism delineates a set of tasks that can be executed simultaneously, beginning at the same starting point, the fork, and continuing until all concurrent tasks are finished having reached the join point. Only when all the concurrent tasks defined by the fork–join have been completed will the succeeding computation proceed.
- Fork–join parallelism is often used to divide instances of a given loop among multiple physical execution resources. This is referred to as "loop parallelism".
- Divide-and-conquer parallelism divides a large problem into two or more smaller problems that can be performed concurrently. Each of the smaller problems may be further subdivided to produce yet more parallel actions of even smaller work. This recursive dividing of work repeatedly into ever smaller subtasks increases the application parallelism until the smallest resulting tasks are trivial to perform.
- Quicksort is an example of a divide-and-conquer algorithm for ordering data.
- The regular sampling parallel sort algorithm improves efficiency and scalability for distributed computing, still borrowing from the quicksort method.
- Manager–worker workflow has one process, the manager, controlling the remaining worker processes, which exhibit the parallelism required to speed up the execution of the total workload. With a central control process, load balancing can be dynamically adapted to evolving data states.
- Embarrassingly parallel algorithms are a subclass of manager–worker algorithms. They are called embarrassingly parallel because the available concurrency is trivially extracted from the workflow.
- A halo is a region exterior to the data subset mapped to a parallel task. It acts as an artificial boundary to that data subset and contains information that originates from the data subsets of neighboring parallel tasks.
- Halo exchange enables each task to perform computations and update the subset of data mapped to that task while having access to any data necessary for such computations that may not be local.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 330

- Sparse matrix calculations exploit arrays (e.g., vectors) that are mostly populated with elements of value zero and where only a relatively small number of the elements are nonzero. Sparse data structures compress the matrix by only storing the nonzero elements, thereby permitting much larger matrices to be represented than the main memory of a computer could otherwise store.
- Task dataflow algorithms represent the precedent constraints among subtasks by their dependencies in the form of a directed acyclic graph. This establishes which tasks must be completed prior to initiating a succeeding task.

9.10 EXERCISES

1. Implement the regular sampling parallel sort algorithm using a message-passing interface (MPI). Plot the time to solution as a function of the number of processes. Include the performance using serial quicksort as a comparison.
2. Compute the Mandelbrot set using MPI with a manager—worker algorithm. Produce a picture of the Mandelbrot set and of the speeding up as a function of the number of processes.
3. Using MPI, write a distributed sparse matrix vector multiplication based on halo exchange of the dense vector. Use the Fluorem/HV15R matrix from the SuiteSparse Matrix Collection [6] and generate a random dense vector. Plot the time to solution of the sparse matrix vector multiplication as a function of the number of processes. Include the memory bandwidth performance for the machine on which you run as given by the HPC Challenge memory bandwidth benchmark.
4. Implement the advection equation using finite difference as illustrated in this chapter using MPI. Plot the solution as a function of time and indicate in the plot which process calculated which point in the solution.
5. Explore the numerical methods identified in the "seven dwarfs". For each numerical method, list the different parallel algorithms that have been historically applied for solving the method. List the reasons that make it difficult to identify the best parallel algorithm for a numerical method.

REFERENCES

[1] P. Colella, Defining Software Requirements for Scientific Computing, 2004.

[2] K. Asanovic, et al., The Landscape of Parallel Computing Research: A View from Berkeley, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, 2006. TR# UCB/EECS-2006-183.

[3] Wikipedia, Quicksort, [Online]. https://en.wikipedia.org/wiki/Quicksort.

[4] M. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw Hill Education, London, 2008 (Chapter 14).

[5] Wikipedia, Cannon's Algorithm, [Online]. https://en.wikipedia.org/wiki/Cannon%27s_algorithm.

[6] SuiteSparse Matrix Collection: Fluorem/HV15R, [Online]. https://www.cise.ufl.edu/research/sparse/matrices/Fluorem/HV15R.html.

CHAPTER

LIBRARIES

10

CHAPTER OUTLINE

10.1 Introduction 313
10.2 Linear Algebra 315
10.2.1 Basic Linear Algebra Subprograms 317
10.2.2 Linear Algebra Package 324
10.2.3 Scalable Linear Algebra Package 326
10.2.4 GNU Scientific Library 326
10.2.5 Supernodal LU 326
10.2.6 Portable Extensible Toolkit for Scientific Computation 327
10.2.7 Scalable Library for Eigenvalue Problem Computations 328
10.2.8 Eigenvalue SoLvers for Petaflop-Applications 328
10.2.9 Hypre: Scalable Linear Solvers and Multigrid Methods 328
10.2.10 Domain-Specific Languages for Linear Algebra 329
10.3 Partial Differential Equations 329
10.4 Graph Algorithms 329
10.5 Parallel Input/Output 330
10.6 Mesh Decomposition 333
10.7 Visualization 334
10.8 Parallelization 334
10.9 Signal Processing 334
10.10 Performance Monitoring 341
10.11 Summary and Outcomes of Chapter 10 342
10.12 Exercises 343
References 344

10.1 INTRODUCTION

Computational science applications use a significant amount of the available high performance computing (HPC) resources. A typical breakdown of the types of computational science research areas represented on HPC resources is presented in Fig. 10.1. This summary of HPC allocations originates from the Extreme Science and Engineering Discovery Environment (XSEDE) virtual system [1], which integrates 12 very large HPC resources for use in peer-reviewed research.

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00010-1
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 332

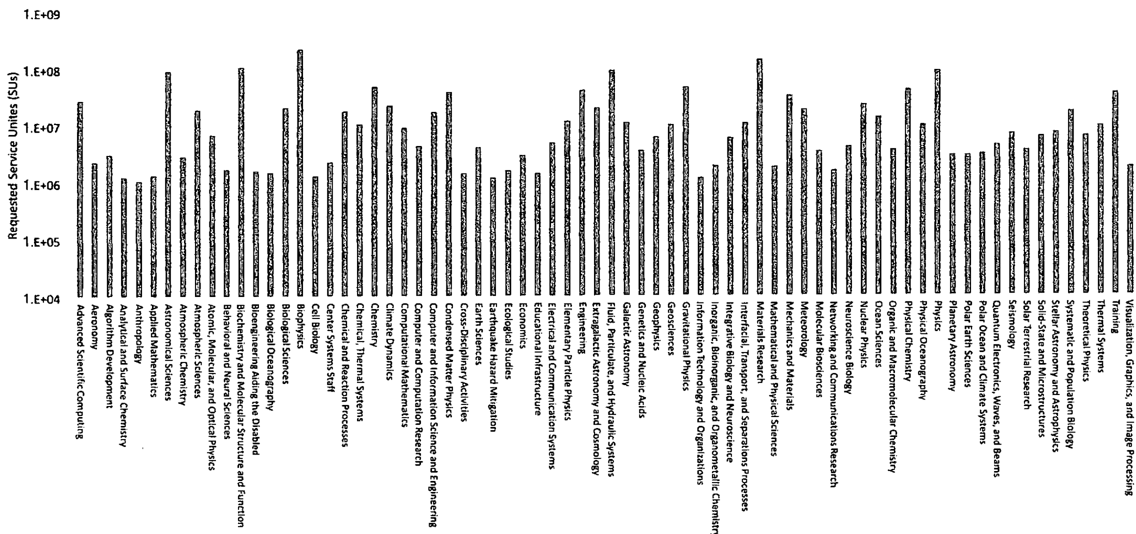


FIGURE 10.1

Extreme Science and Engineering Discovery Environment (XSEDE) allocation summary for 2015 indicating the number of service units (SUs) allocated to various research areas. Only research areas with allocations exceeding 1 million SUs are listed here. An SU is defined locally on each supercomputer, but is generally the walltime in hours multiplied by the number of cores used for a simulation. Thus a simulation requiring 64 cores that ran for 1 h would be charged 64 SUs. The research areas receiving the most HPC time from XSEDE in 2015 were biophysics and materials research.

While these applications are used in a wide variety of very different disciplines, their underlying computational algorithms are frequently very similar to one another. As a consequence, several software libraries have been developed for HPC resources to fill a specific computing need, so application developers do not have to waste time redeveloping supercomputing software that has already been developed elsewhere. Subsequently, these libraries end up becoming required software dependencies across many user applications, and their performance and usage become critically important for an application's performance. Libraries targeting numerical linear algebra operations are the most common, given the ubiquity of linear algebra in scientific computing algorithms. Other libraries target operations like input/output (I/O), fast Fourier transform (FFT), the finite element method, and solving ordinary differential equations. These libraries have generally been highly tuned for performance, often for more than a decade, making it difficult for the casual application developer to match a library's performance using a homemade equivalent. On account of their ease of use and their highly tuned performance across a wide range of HPC platforms, the use of scientific computing libraries as software dependencies in computational science applications has become widespread.

Apart from acting as a repository for software reuse, libraries serve the important role of providing a knowledge base for specific computational science domains. These libraries become community standards and serve as ways for members of the community to communicate with one another. This chapter explores some of the most widely used libraries in computational science and their characteristics on HPC resources. An abbreviated list of some of the most important libraries for scientific computing is found in Table 10.1. Each of the application domains in Table 10.1 is explored in the following sections.

10.2 LINEAR ALGEBRA

Numerical linear algebra is a key component to a large number of HPC applications, and libraries that provide numerical algorithms for solving sets of linear equations are among the most widely used on modern supercomputers. This is illustrated in part in Fig. 10.2, where a small sample of widely used

Table 10.1 Some Widely Used Libraries on HPC Systems and Their Associated Application Domains

Application Domain	Widely Used Libraries on HPC Systems
Linear algebra	BLAS [2], Lapack [3], ScaLapack [4], GNU Scientific Library [5], SuperLU [6], PETSc [7], SLEPc [8], ELPA [9], Hypre [10]
Partial differential equations	PETSc [7], Trilinos [11]
Graph algorithms	Boost Graph Library [12], Parallel Boost Graph Library [12]
Input/output	HDF5 [13], Netcdf [14], Silo [15]
Mesh decomposition	METIS [16], ParMETIS [17]
Visualization	VTK [18]
Parallelization	Pthreads, MPI, Boost MPI [12]
Signal processing	FFTW [19]
Performance monitoring	PAPI [20], Vampir [21]

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 334

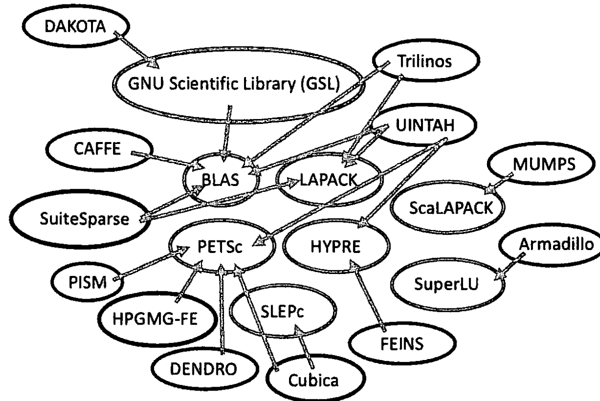


FIGURE 10.2

A small sample of core linear algebra libraries (blue [dark gray in print versions]) and a small sample of widely used application frameworks with dependencies on these libraries (red [black in print versions]). The dependencies (sometimes optional) are indicated by *green arrows* (light gray in print versions). The most fundamental libraries, basic linear algebra subprograms (BLAS), the linear algebra package (Lapack), and the portable, extensible toolkit for scientific computation (PETSc), show up very frequently as application dependencies. The application frameworks represented here include the Dakota software toolkit [22], the Caffe deep-learning framework [23], the SuiteSparse suite of sparse matrix algorithms [24], the parallel ice sheet model [25], the finite element high performance geometric multigrid benchmark [26], the Dendro suite of parallel algorithms [27], the Cubica toolkit for subspace deformations [28], the finite element incompressible Navier–Stokes solver [29], the Armadillo C++ linear algebra library [30], the multifrontal massively parallel sparse direct Solver [31], the UINTAH software suite [32], and the Trilinos project [11].

application frameworks is listed along with their dependencies on some of the key linear algebra libraries explored in this chapter. In addition to application frameworks, numerical linear algebra is a principal component of many of the key HPC benchmarks. For instance, seven of the HPC benchmarks explored in Chapter 3 deal with numerical linear algebra performance—highly parallel Linpack, DGEMM, high performance conjugate gradients, conjugate gradient, BT, SP, and lower/upper (LU)—reflecting the impact this discipline has on HPC.

This section explores several types of numerical linear algebra libraries, including very low abstraction level serial libraries like BLAS [2], higher abstraction level parallel libraries with extensive sparse matrix support like PETSc [7], and very high abstraction level domain-specific language libraries like MTL4 [33] and Blaze [34].

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 335

10.2.1 BASIC LINEAR ALGEBRA SUBPROGRAMS

Photo by Pierre Lescanne via Wikimedia Commons

John Backus was the cocreator of the first practical and widely used computer programming language, Fortran. In 1953 he assembled and led a team of 10 researchers at IBM whose task was to find an approach that would simplify the programming of computers while permitting proper structuring of the executable code to make it more understandable to other programmers. In times when computers were predominantly coded in machine language targeting a specific architecture that demanded a thorough understanding of machine internals, this was a truly groundbreaking development. The Fortran language, short for formula translator, was released in 1957 and combined elements of algebra and English language. This high-level language and its compiler (originally written in 25,000 lines of code) enabled practical portability and platform independence of computer programs. While Fortran syntax and concepts have been updated several times since its inception, it remains one of the most common programming languages in supercomputing and has an extensive set of software libraries supporting many domains of computational science.

John Backus is also known for developing the Backus–Naur Form (BNF), a metalanguage for expressing context-free grammars. For this contribution he was honored with the Association for Computing Machinery Turing Award in 1977. BNF is commonly used to describe the syntax of various programming languages, communication protocols, file formats, and others. Backus helped develop the influential ALGOL programming language that introduced many important procedural programming concepts; the original ALGOL variant has been fleshed out in BNF. His later work on the FP language and its descendant FL inspired broader research in functional programming.

For his achievements, John Backus was awarded an IBM Fellowship in 1963; he also received a National Medal of Science in 1975, the Harold Pender Award in 1983, and the Charles Stark Draper Prize in 1993.

BLAS provides a standard interface to vector, matrix–vector, and matrix–matrix routines that have been optimized for various computer architectures. In addition to the reference implementation [2], which provides both Fortran 77 and C interfaces, and the Automatically Tuned Linear Algebra Software project [35], which also has a BLAS implementation, there are multiple vendor-provided BLAS libraries optimized for their respective hardware. Finally, the Boost libraries [12] provide a C++ template class with BLAS functionality called uBLAS.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 336

BLAS design and implementation was handled by Charles Lawson, Richard Hanson, F. Krogh, D.R. Kincaid, and Jack Dongarra beginning in the 1970s; the genesis of the idea for BLAS is credited to Lawson and Hanson while they were working at NASA's Jet Propulsion Laboratory [36]. BLAS development coincided with development of the Linpack package introduced in Chapter 3. Linpack was the first major package to incorporate the BLAS library.

The first BLAS routines developed were limited to vector operations, including inner products, norms, adding vectors, and scalar multiplication, and are typified by the operations of Eq. (10.1),

$$y = \alpha x + y \tag{10.1}$$

where x, y are vectors and α is a scalar value. These vector–vector operations are referred to as BLAS Level 1. At the time the fastest supercomputer in the world was the Control Data Corporation (CDC)-7600 (shown in Fig. 10.3), which had such a small cache size that matrix operations were not possible, thereby limiting the first BLAS routines to vector operations. The CDC-7600 further motivated the BLAS creators to focus on developing a portable linear algebra interface so that others would not have to compile assembly code by hand to utilize the CDC-7600's capabilities fully.

In 1987, about 10 years after BLAS Level 1 was released, routines for matrix–vector operations became available, followed by matrix–matrix operations in 1989. These later additions are the Level 2 (matrix–vector) and Level 3 (matrix–matrix) BLAS operations, typified by Eqs. (10.2)–(10.3).

$$y = \alpha Ax + \beta y \tag{10.2}$$

$$C = \alpha AB + \beta C \tag{10.3}$$

FIGURE 10.3

A section of the CDC-7600. The CDC-7600 could achieve up to 36 Mflops and was the fastest computer available from 1969 to 1975.

Photo by Jitze Couperus via Wikimedia Commons

BLAS matrix–vector operations are illustrated in Eq. (10.2), where x and y are vectors and α and β are scalars. BLAS matrix–matrix operations are illustrated in Eq. (10.3), where A, B, and C are matrices and α and β are scalars.

Each routine in BLAS has a specific naming convention that specifies the precision of the operation, the type of matrix (if any) involved, and the operation to perform. BLAS is natively written in Fortran 77, but C bindings to BLAS are available via CBLAS and are used in this chapter for illustration. For BLAS Level 1 operations there is no matrix involved and so the naming convention for each routine begins with *cblas_* after which a precision prefix is placed before the operation name. The core BLAS precision prefixes are summarized in Table 10.2. While these are the core precision prefixes, some BLAS operations support mixed precisions, resulting in combinations of the listed prefixes.

BLAS Level 1 operations can be subdivided into four different subgroups: vector rotations (Table 10.3), vector operations without a dot product (Table 10.4), vector operations with a dot product (Table 10.5), and vector norms (Table 10.6).

BLAS Level 2 and Level 3 operations involve matrices, and indicate the type of matrix they support in their name. Levels 2 and 3 names are of the form *cblas_*pmmoo, where the p indicates the precision, mm indicates the matrix type, and oo indicates the operation. Possible matrix types are listed in Table 10.7. Apart from general matrices, all other matrix types come in three storage scheme flavors: dense (default), banded (indicated by a "b" in the name), and packed (indicated by a "p" in the name). Dense storage schemes are either row-based or column-based storage in a continuous memory array. Packed storage schemes hold matrix values that are packed by rows or columns in a one-dimensional array, while band storage is applied to sparse matrices where the nonzero entries lie in diagonal bands. An example of a banded matrix is a tridiagonal matrix which has nonzero column entries at the $i - 1$, i, and $i + 1$ columns for the *ith* row. In band storage for a banded matrix, the diagonal bands to the left of the main diagonal ("subdiagonals") and diagonal bands to the right of the diagonal ("superdiagonals") are placed in a two-dimensional (2D) array.

BLAS Levels 2 and 3 operations are summarized in Table 10.8.

As an example, the name of the BLAS Level 3 routine *cblas_dgemm* indicates that this routine will perform a double-precision dense matrix–matrix multiplication. DGEMM is also the name for the matrix–matrix multiplication benchmark in the HPC Challenge suite introduced in Chapter 4.

Table 10.2 Precision Prefixes Used by BLAS Routines

Prefix	Description
s	Single precision (float), 4 bytes
d	Double precision (double), 8 bytes
c	Complex (two floats), 8 bytes
z	Complex*16 (two doubles), 16 bytes

Some BLAS operations support mixed precision operations resulting in combinations of the following prefixes as well.

Table 10.3 BLAS Level 1 Rotation Operations

Name	Description	Supported Precisions
rotg	Computes parameters for a Givens rotation; that is, given scalars a and b, compute c and s so that $\begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$ where $r = \sqrt{\lvert a\rvert^2 + \lvert b\rvert^2}$	s,d
rot	Applies the Givens rotation; that is, provided two vectors as input, x and y, each vector element is replaced as follows: $x_i = cx_i + sy_i$ $y_i = -sx_i + cy_i$ where c and s are the parameters for the Givens rotation (see rotg)	s,d
rotmg	Computes the 2×2 modified Givens rotation matrix $H = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix}$ That is, given scaling factors d_1 and d_2 with Cartesian coordinates (x_1,y_1) of an input vector, compute the modified Givens rotation matrix H such that $\begin{pmatrix} x_1 \\ 0 \end{pmatrix} = H\begin{pmatrix} x_1\sqrt{d_1} \\ y_1\sqrt{d_2} \end{pmatrix}$	s,d
rotm	Applies the modified Givens rotation; that is, provided two vectors, x and y, compute: $\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix}\begin{pmatrix} x_i \\ y_i \end{pmatrix}$ where h_{ij} are the elements of the modified Givens rotation matrix (see rotmg)	s,d

Table 10.4 BLAS Level 1 Vector Operations Without a Dot Product

Name	Description	Supported Precisions
swap	Swaps vectors $x \leftrightarrow y$	s,d,c,z
scal	Scales a vector by a constant $y = \alpha y$	s,d,c,z,cs,zd
copy	Copies a vector $y = x$	s,d,c,z
axpy	Updates a vector $y = \alpha x + y$	s,d,c,z

Note that the scal operation supports mixed precisions, where the α single-precision or double-precision constant can be multiplied by a complex vector.

Table 10.5 BLAS Level 1 Vector Operations Involving a Dot Product

Name	Description	Supported Precisions
dot	Dot product $x^T y$	s,d,ds
dotc	Complex conjugate dot product $x^h y$	c,z
dotu	Complex dot product $x^T y$	c,z
sdsdot	Dot product plus a scalar $\alpha + x^T y$	sds

Table 10.6 BLAS Level 1 Vector Operations Involving a Norm

Name	Description	Supported Precisions
nrm2	Compute the 2-norm $\|x\|_2 = \sqrt{\sum \|x_i\|^2}$	s,d,sc,dz
asum	Compute the 1-norm $\|x\|_1 = \sum \|x_i\|$	s,d,sc,dz
i_amax	Compute the ∞-norm $\|x\|_\infty = \max(\|x_i\|)$	s,d,c,z

Table 10.7 Matrix Types Supported in BLAS Levels 2 and 3

Matrix Type	Description
General: ge,gb	General, nonsymmetric, possibly rectangular matrix.
Symmetric: sy,sb,sp	Symmetric matrix. This is a special class of square matrix that is equal to its own transpose. So for matrix A with elements a_{ij}, a symmetric matrix would have elements which satisfy $a_{ij} = a_{ji}$.
Hermitian: he,hb,hp	Hermitian matrix. This is a special class of square matrix that is equal to its own Hermitian conjugate. So for matrix A with elements a_{ml}, matrix A is Hermitian if all elements satisfy $a_{ml} = \overline{a_{lm}}$ where the overbar is the complex conjugate.
Triangular: tr,tb,tp	Triangular matrix. This is a special class of square matrices where all the entries above the diagonal are zero (lower triangular) or all the entries below the diagonal are zero (upper triangular).

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 340

Table 10.8 BLAS Levels 2 and 3 Operations

Name	Description
mv	Matrix–vector product
sv	Solve matrix (only for triangular matrices)
mm	Matrix–matrix product, $C = \alpha AB + \beta C$ where A, B, C are matrices and α, β are scalars
rk	Rank-k update, $C = \alpha A A^T + \beta C$ where A, C are matrices and α, β are scalars
r2k	Rank-2k update, $C = \alpha AB^T + \overline{\alpha} BA^T + \beta C$ where A, B, C are matrices and α, β are scalars

The *cblas_dgemm* routine takes 14 arguments, shown here:

```
void cblas_dgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const double alpha, const double *A,
                 const int lda, const double *B, const int ldb,
                 const double beta, double *C, const int ldc);
```

- *Order* indicates the storage layout as either row major or column major. This input is either *CblasRowMajor* or *CblasColMajor*.
- *TransA* indicates whether to transpose matrix *A*. This input is either *CblasNoTrans*, *CblasTrans*, or *CblasConjTrans*, indicating no transponse, transpose, or complex conjugate transpose, respectively.
- *TransB* indicates whether to transpose matrix *B*. Acceptable options are the same as those listed for *A*.
- *M* indicates the number of rows in matrices *A* and *C*.
- *N* indicates the number of columns in matrices *B* and *C*.
- *K* indicates the number of columns in matrix *A* and the number of rows in matrix *B*. This is the shared index between matrices *A* and *B*.
- *alpha* is the scaling factor for *A***B*.
- *A* is the pointer to matrix *A* data.
- *lda* is the size of the first dimension of matrix *A*.
- *B* is the pointer to matrix *B* data.
- *lbd* is the size of the first dimension of matrix *B*.
- *beta* is the scaling factor for matrix *C*.
- *C* is the pointer to matrix *C* data.
- *ldc* is the size of the first dimension of matrix *C*.

An example of matrix–matrix multiplication is provided in Fig. 10.4. In this example a 3×3 matrix–matrix product is computed.

$$\begin{pmatrix} 47.1 & 56.52 & 65.94 \\ 131.88 & 169.56 & 207.24 \\ 216.66 & 282.6 & 348.54 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 0 & 3.14 & 6.28 \\ 9.42 & 12.56 & 15.7 \\ 18.84 & 21.98 & 25.12 \end{pmatrix}$$

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 341

```
0001 #include <stdio.h>
0002 #include <stdlib.h>
0003 #include <cblas.h>
0004
0005 int main()
0006 {
0007   double *A, *B, *C;
0008   int m = 3; // square matrix, number of rows and columns
0009   int i,j;
0010
0011   A = (double *) malloc(m*m*sizeof(double));
0012   B = (double *) malloc(m*m*sizeof(double));
0013   C = (double *) malloc(m*m*sizeof(double));
0014
0015   // initialize the matrices
0016   for (i=0;i<m;i++) {
0017     for (j=0;j<m;j++) {
0018       A[j + m*i] = j + m*i;  // arbitrarily initialized
0019       B[j + m*i] = 3.14*(j + m*i);
0020       C[j + m*i] = 0.0;
0021     }
0022   }
0023   double alpha = 1.0;
0024   double beta = 0.0;
0025
0026   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
0027               m, m, m, alpha, A, m, B, m, beta, C, m);
0028
0029   for (i=0;i<m;i++) {
0030     for (j=0;j<m;j++) {
0031       printf(" A[%d][%d]=%g ",i,j,A[j+m*i]);
0032     }
0033     printf("\n");
0034   }
0035
0036   for (i=0;i<m;i++) {
0037     for (j=0;j<m;j++) {
0038       printf(" B[%d][%d]=%g ",i,j,B[j+m*i]);
0039     }
0040     printf("\n");
0041   }
0042
0043   for (i=0;i<m;i++) {
0044     for (j=0;j<m;j++) {
0045       printf(" C[%d][%d]=%g ",i,j,C[j+m*i]);
0046     }
0047     printf("\n");
0048   }
0049
0050
0051   free(A);
0052   free(B);
0053   free(C);
0054   return 0;
0055 }
```

FIGURE 10.4

Example of multiplying two 3 × 3 matrices using *cblas_dgemm*.

10.2.2 LINEAR ALGEBRA PACKAGE

Lapack [3] was developed by a collaboration between Jack Dongarra, James Demmel, and others, and provides driver routines designed to solve complete problems such as a system of linear equations, eigenvalue problems, and singular value problems. It also provides computational routines that can perform specific tasks like LU or Cholesky factorization. Certain auxiliary routines are provided for common subtasks. Lapack requires BLAS Level 2 and Level 3 functionality, and it supersedes the Linpack library. Unlike Linpack, which also required BLAS but which targeted vector machines with shared memory, Lapack is designed around the cache-based memory hierarchies found on modern supercomputers. It was initially written in Fortran 77, but switched to Fortran 90 in 2008. A C interface to Lapack is provided by using Lapacke [37].

The naming scheme for Lapack routines is similar to BLAS. All routines are in the form of XYYZZZ, where X is the data type (one of s, d, c, or z, as in Table 10.2), YY is the type of matrix, and ZZZ is the computation performed. Lapack matrix types share all the BLAS matrix types in Table 10.7 and use the same names. Lapack has some additional matrix types, including unitary matrices and symmetric positive definite matrices among others. Like BLAS, Lapack provides support for dense, banded, and packed storage formats, but not for general sparse matrices. Driver routines are summarized in Table 10.9. Expert versions of some of these drivers are available by appending an x to the name; these versions provide more functionality but also generally require more memory. In some cases multiple driver routines are available to solve the same problem type reflecting different underlying algorithms.

Table 10.9 Lapack Driver Routines

Driver Name	Description
SV	Solver for system of linear equations: $Ax = b$
LS, LSY, LSS, LSD	Solver for linear least squares problems: minimize x in$\|b - Ax\|_2$ where A is not necessarily a square matrix, generally with more rows than columns as would occur in an overdetermined system of linear equations.
LSE	Linear equality-constrained least squares problems: minimize x in $\|c - Ax\|_2$ subject to the constraint that $Bx = d$ where A is an $m \times n$ matrix, c is a vector of size m, B is a $p \times n$ matrix, and d is a vector of size p, where $p \leq n \leq m + p$.
GLM	General linear model problems: minimize x in $\|y\|_2$ subject to the constraint that $d = Ax + By$ where A is a $m \times n$ matrix, B is a $n \times p$ matrix, d is a vector of size n, and $m \leq n \leq m + p$.
EV, EVD, EVR	Symmetric eigenvalue problems: find eigenvalues λ and eigenvectors k where $Ak = \lambda k$ for a symmetric matrix A.
ES	Nonsymmetric eigenvalue problems: find eigenvalues λ and eigenvectors k where $Ak = \lambda k$ for a nonsymmetric matrix A.
SVD, SDD	Compute the singular value decomposition of $m \times n$ matrix A: $A = UDV^T$ where matrices U and V are orthogonal and D is a diagonal real matrix of size $m \times n$ containing the singular values of matrix A.

Some of these drivers also comes in an expert flavor accessible by appending an X to the name. The expert flavor provides additional functionality but also generally requires more memory. In some cases, multiple drivers are available to solve the same problem type using different algorithms.

While Lapack is written in Fortran, C bindings are available through the Lapacke library which comes with Lapack. Fortran routines in Lapack can be called directly from C code, but the C bindings simplify code portability. The naming convention for Lapacke remains the same as Lapack but prefixes *LAPACKE_* to each routine. An example solving a system of linear equations in double precision is given in Fig. 10.5. This example solves the linear system:

$$\begin{pmatrix} 1 & 3 & 2 \\ 4 & 1 & 9 \\ 5 & 7 & 2 \end{pmatrix} x = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}$$

There are eight arguments to the dgesv routine:

```
lapack_int LAPACKE_dgesv( int matrix_layout, lapack_int n, lapack_int nrhs,
                          double* a, lapack_int lda, lapack_int* ipiv,
                          double* b, lapack_int ldb );
```

```
#include <stdio.h>
#include <lapacke.h>

int main (int argc, const char * argv[])
{
   double A[3][3] = {1,3,2,4,1,9,5,7,2};
   double b[3] = {-1,-1,1};
   lapack_int ipiv[3];
   lapack_int info,m,lda,ldb,nrhs;
   int i,j;

   m = 3;
   nrhs = 1;
   lda = 3;
   ldb = 1;

   // Solve the linear system
   info = LAPACKE_dgesv(LAPACK_ROW_MAJOR,m,nrhs,*A,lda,ipiv,b,ldb);

   // check for singularity
   if (info > 0 ) {
     printf(" U(%d,%d) is zero! A is singular\n",info,info);
     return 0;
   }

   // print the answer
   for (i=0;i<m;i++) {
     printf(" b[%i] = %g\n",i,b[i]);
   }

   printf( "\n" );
   return 0;
}
```

FIGURE 10.5

Example of Lapack DGESV general matrix solve $(Ax = b)$ with one right hand side vector, b. Here the C bindings to Lapack (Lapacke) are used.

- *matrix_layout* specifies the whether the matrix comes in row-major or column-major form. Acceptable inputs are either LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.
- *n* indicates the size of the square matrix.
- *nrhs* indicates the number of right-hand-side vectors on which to perform the solve. dgesv can solve multiple right-hand sides in each call.
- *a* is the matrix.
- *lda* is the size of the first dimension of the matrix.
- *ipiv* is a vector of size *n* containing the pivot points.
- *b* is the right-hand-side vector.
- *ldb* is the size of the first dimension of the right-hand-side vector.

10.2.3 SCALABLE LINEAR ALGEBRA PACKAGE

The scalable linear algebra package (ScaLapack) [4] is the HPC equivalent of Lapack and shares much of the same interface. It is built on message passing, and relies on a parallel version of BLAS called PBLAS that accompanies the library. The relationship between ScaLapack and PBLAS is analogous to the dependency of Lapack on BLAS Levels 2 and 3 routines. Like Lapack, support is available for dense and banded matrices but not general sparse matrices. Matrices are decomposed in a 2D block-cyclic distribution across processes for use on distributed-memory architectures. The 2D block-cyclic distribution decomposes the matrix into 2D blocks of size $m_{block} \times n_{block}$ which are then mapped on to the processes.

10.2.4 GNU SCIENTIFIC LIBRARY

The GNU scientific library (GSL) [5] provides a wide array of linear algebra routines, including an interface to BLAS for C and C++. Unlike the other libraries described so far, support for general sparse matrices is provided in GSL, along with support iterative solvers for sparse systems of linear equations.

As an example of the GSL interface to dgemm is shown in Fig. 10.6.

10.2.5 SUPERNODAL LU

Supernodal LU (SuperLU) [6] is a library for direct solves of general sparse systems of equations through LU decomposition on HPC systems. It supports shared-memory and distributed-memory architectures as well as accelerator architectures such as graphics processing units (GPUs). Like Lapack and ScaLapack, it can solve multiple right-hand-side vectors in a single call for improved efficiency. The right-hand-side vectors are assumed to be dense, while the matrix must be square and is assumed to be sparse (dominated by zero entries). SuperLU consists of three libraries:

- Sequential SuperLU, like Lapack, is designed for sequential execution on processors with cache-based memory hierarchies.
- Multithreaded SuperLU is designed for SMP architectures.
- Distributed SuperLU is designed for distributed-memory architectures. Some routines in this library support hybrid computer architectures incorporating multiple GPUs.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 345

```
0001 #include <stdio.h>
0002 #include <gsl/gsl_blas.h>
0003
0004 int main (void) {
0005   double a[] = { 0,1,2,
0006                  3,4,5,
0007                  6,7,8 };
0008
0009   double b[] = { 0,    3.14, 6.28,
0010                  9.42, 12.56,15.7,
0011                  18.84,21.98,25.12 };
0012
0013   double c[] = { 0.00, 0.00, 0.00,
0014                  0.00, 0.00, 0.00,
0015                  0.00, 0.00, 0.00 };
0016
0017   gsl_matrix_view A = gsl_matrix_view_array(a, 3, 3);
0018   gsl_matrix_view B = gsl_matrix_view_array(b, 3, 3);
0019   gsl_matrix_view C = gsl_matrix_view_array(c, 3, 3);
0020
0021   // Compute C = A B
0022
0023   gsl_blas_dgemm (CblasNoTrans, CblasNoTrans,
0024                   1.0, &A.matrix, &B.matrix,
0025                   0.0, &C.matrix);
0026
0027   printf (" %g, %g, %g\n", c[0], c[1],c[2]);
0028   printf (" %g, %g, %g\n", c[3], c[4],c[5]);
0029   printf (" %g, %g, %g\n", c[6], c[7],c[8]);
0030
0031   return 0;
0032 }
```

FIGURE 10.6

An example of using the BLAS dgemm routine in GSL. The example from Fig. 10.4 is redone here using GSL. The interface to dgemm in GSL simplifies things considerably; the number of arguments is only 7 instead of 14.

SuperLU complements ScaLapack, in that it provides a high performance direct solver for general systems of sparse linear equations whereas ScaLapack provides high performance direct solver support for dense and banded systems of linear equations.

10.2.6 PORTABLE EXTENSIBLE TOOLKIT FOR SCIENTIFIC COMPUTATION

PETSc [7] was started in 1991 and led by William Gropp, with the goal of providing a suite of data structures and routines to aid application scientists in solving partial differential equations on HPC resources. As the discretization of partial differential equations often results in a very large system of sparse linear equations, PETSc provides a large suite of parallel iterative linear equation solvers. These solvers are principally Krylov subspace solvers like the generalized minimum residual (GMRES) method and CG. PETSc also provides simple interfaces for application-specific linear solver preconditioners, including domain decomposition type preconditions like additive Schwartz

Table 10.10 A Small Sample of Distributed Vector Operations in PETSc

Vector Function Name	Description
VecAXPY(Vec y, PetscScalar alpha, Vec x)	$y = \alpha x + y$
VecAYPX(Vec y, PetscScalar alpha,Vec x)	$y = x + \alpha y$
VecPointwiseMult(Vec w, Vec x, Vec y)	$w_i = x_i * y_i$
VecMax(Vec x, PetscInt *p, PetscReal *r)	Returns the maximum value, $r = \max(x_i)$, and its location
VecCopy(Vec x,Vec y)	$y = x$
VecShift(Vec x, PetscScalar s)	$x_i = s + x_i$
VecScale(Vec x, PetscScalar alpha)	$x = \alpha x$

type and others. PETSc provides support for distributed matrices and vectors where each process locally owns a subvector of contiguous data. Selected distributed vector operations in PETSc are listed in Table 10.10. PETSc employs message-passing interface (MPI) for communication on distributed-memory architectures.

PETSc interfaces with a large number of other widely used libraries and forms one of the core libraries found on a supercomputer. Libraries which interface with PETSc include Hypre [38], SLEPc [8], Uintah [32], Sundials [39], Trilinos [11], SuperLU [6], SAMRAI [40], and TAU [41]. An application using PETSc was awarded the Gordon Bell Prize in 1999 [42].

10.2.7 SCALABLE LIBRARY FOR EIGENVALUE PROBLEM COMPUTATIONS

The scalable library for eigenvalue problem computations (SLEPc) [8] is an extension of PETSc and complements ScaLapack in providing an HPC library for solving very large sparse eigenvalue problems with both real and complex numbers. Like PETSc, it is built on the MPI library and shares much in common with PETSc. SLEPc is similar in function to the Fortran 77-based ARPACK software [43], which is also designed to solve large eigenvalue problems using message passing. SLEPc provides a transparent interface to ARPACK.

10.2.8 EIGENVALUE SOLVERS FOR PETAFLOP-APPLICATIONS

For many scientific computing applications such as quantum chemistry, computing the eigenvalues and eigenvectors of Hermitian matrices is a key computational kernel. The Eigenvalue SoLvers for Petaflop-Applications (ELPA) [9] created by the ELPA consortium is free software designed for highly scalable eigenvalue and eigenvector computations on Hermitian matrices. ELPA uses BLAS, Lapack, the basic linear algebra communication subroutines [44], ScaLapack, and MPI. ELPA is widely used in the materials science community on HPC resources via the density functional theory toolkit VASP [45].

10.2.9 HYPRE: SCALABLE LINEAR SOLVERS AND MULTIGRID METHODS

The Hypre library [38] provides a set of highly scalable preconditioners for systems of linear equations, as well as scalable iterative solvers and algebraic multigrid algorithms that have found broad

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 347

usage in the HPC community. Hypre uses MPI for communication and interfaces with the PETSc library. Like PETSc, it also provides support for distributed vectors and matrices.

10.2.10 DOMAIN-SPECIFIC LANGUAGES FOR LINEAR ALGEBRA

The complexity of using linear algebra library routines like those in BLAS, Lapack, or PETSc has motivated in part the development of several higher-level abstraction interfaces so that application developers can develop distributed linear algebra applications using code that is very simple to read. The MATLAB® framework [46] is a proprietary example of such an approach, but is not competitive in terms of performance with the libraries presented in this section. A template library which achieves comparable performance with PETSc for sparse linear algebra operations but retains the look and feel of the original mathematical notation of linear algebra is MTL4 [33]. An example of MTL4 is shown in Fig. 10.7: it creates a Laplacian matrix, computes a sparse matrix−vector multiplication, and then performs a linear solve using a Krylov solver. The output from this code is shown in Fig. 10.8. The MPI distributed-memory version of the example MTL4 code in Fig. 10.7 is shown in Fig. 10.9. Another library with a similar goal to MTL4 is Blaze [34]. These two are a small sample of the many libraries available that aim to address the growing need in linear algebra libraries for both HPC capability and an intuitive interface to simplify application development.

10.3 PARTIAL DIFFERENTIAL EQUATIONS

PETSc [7], mentioned in Section 10.2.6 is one of the most important toolkits for solving systems of partial differential equations. Beyond supporting distributed vectors and matrices as well as distributed Krylov subspace methods like GMRES and CG, PETSc provides ordinary differential equation integrators and nonlinear solvers, including Newton-based methods.

A second widely used library for solving systems of partial differential equations is the Trilinos project [11]. Trilinos is a collection of libraries spread across 10 different capability areas, each with a direct impact on applications targeting the solution of partial differential equations. These capability areas range from the standard scalable linear algebra support to nontraditional parallel programming environments to provide portability across multiple HPC architectures while leveraging architecture-dependent system capabilities.

10.4 GRAPH ALGORITHMS

Sparse graph algorithms such as the breadth first search explored in Chapter 9 form a crucial component in many core HPC algorithms, such as shortest path problems, PageRank, and network flow problems. Three libraries available for high performance graph algorithms are the Parallel Boost Graph Library (PBGL) [12], Combinatorial BLAS [47], and Giraph [48]. PBGL extends the Boost Graph Library for HPC and provides a large number of graph algorithms for distributed-memory architectures. Combinatorial BLAS is another parallel graph library which provides linear algebra primitives for graphs and also targets distributed-memory architectures.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 348

```
0001 #include <iostream>
0002 #include <boost/numeric/mtl/mtl.hpp>
0003 #include <boost/numeric/itl/itl.hpp>
0004
0005
0006 int main(int argc, char* argv[])
0007 {
0008     using namespace mtl;
0009
0010     mtl::par::environment      env(argc, argv);
0011
0012     // Use compressed sparse row format for sparse matrix element storage
0013     typedef matrix::compressed2D<double>    matrix_type;
0014
0015     typedef mtl::vector::dense_vector<double>   vector_type;
0016
0017     matrix_type A;
0018
0019     int n = 100;
0020     laplacian_setup(A,n,n);
0021
0022     vector_type x(num_rows(A),1.0),b;
0023
0024     // Sparse matrix vector multiplication
0025     b = A * x;
0026
0027     // Compute the two norm
0028     double mbnorm = two_norm(b);
0029     printf(" b vector l2norm %10.2f\n",mbnorm);
0030
0031     // reset x vector to be zero
0032     x= 0;
0033
0034     // Solve for x in Ax=b using a Krylov solver, BiCGStabilized.
0035     // Use the ILU_0 preconditioner
0036     itl::pc::ilu_0<matrix_type>     P(A);
0037     itl::cyclic_iteration<double> iter(b, 500, 1.e-8, 0.0, 5);
0038     bicgstab_2(A, x, b, P, iter);
0039
0040     // Print an element of x (should be one)
0041     printf(" x[1] = %g (should be one)\n",x(1));
0042
0043     return 0;
0044 }
```

FIGURE 10.7

A sparse linear algebra example using MTL4. This code stores a matrix in compressed sparse rows format (line 13), creates a Laplacian matrix (line 20), creates two vectors (a and b) (line 22), initializes vector x to be one (line 22), computes the sparse matrix vector product of A∗x (line 25), resets x to be zero (line 32), and solves Ax=b (line 38).

10.5 PARALLEL INPUT/OUTPUT

Parallel I/O libraries provide high performance output to a single file to avoid the problems associated with nonparallel I/O, including poor performance and creating a large number of individual files each

```
[ b vector l2norm      20.20
iteration 0: resid 13.0643
iteration 5: resid 0.272981
iteration 10: resid 0.11331
iteration 15: resid 0.00256046
iteration 20: resid 4.89401e-05
iteration 25: resid 4.90882e-06
finished! error code = 0
26 iterations
7.61006e-08 is actual final residual.
3.76754e-09 is actual relative tolerance achieved.
Relative tol: 1e-08  Absolute tol: 0
Convergence:  0.474244
 x[1] = 1 (should be one)
```

FIGURE 10.8

Output from the MTL4 example in Fig. 10.7.

written by a single process that must be combined in postprocessing. Common libraries used for HPC I/O include the Network Common Data Form (NetCDF) [49] and the Hierarchical Data Format (HDF5) [50].

NetCDF is a portable format to represent scientific data and has been used extensively in climate modeling, satellite data processing, and geological institutes. NetCDF files are self-describing, portable across hardware architectures, and directly appendable. However, one of the most appealing properties of this data form is that it is archivable, meaning backward compatibility with earlier versions of NetCDF data is supported.

The HDF library was first created in 1988 at the National Center for Supercomputing Applications at the University of Illinois at Urbana–Champaign and, like NetCDF, provides a self-describing, portable data format. HDF5 is the most recent version of the format and provides support for parallel I/O. HDF5 parallel I/O is built on top of the MPI I/O functionality. An example using the HDF5 library to write an array of particle data to a file in HDF5 format is provided in Fig. 10.10.

The HDF5 library also provides a series of tools for examining HDF5 format data, including the tools *h5ls* and *h5dump*. *h5ls* is analogous to the Unix *ls* command and enables the user to query the HDF5 namespace in the same way *ls* queries the Unix file system directory. Executing *h5ls* on the "particles.h5" output file produced in Fig. 10.10 results in the following output:

Particle/data Dataset{15}

The *h5dump* utility will dump to screen the data stored in the hdf5 file. The small portion of output resulting from executing *h5dump* on the "particles.h5" file in Fig. 10.10 is shown in Fig. 10.11.

The Silo library [15] developed at Lawrence Livermore National Laboratory uses lower-level I/O libraries such as HDF5 and portable binary database [51] to simplify implementation of parallel I/O schemes and output for scientific computing applications. Its application programming interface (API) supports output types common to scientific computing, including adaptive mesh refinement and unstructured grids in both 2D and 3D. As an example of simple parallel I/O of a 2D structured unigrid quad mesh, Fig. 10.12 shows how one might use Silo for distributed output. In this example each MPI process holds a local 2D mesh for output, but the number of I/O ranks can be varied by the user. If the

```
#include <iostream>
#include <boost/mpi.hpp>
#include <boost/numeric/mtl/mtl.hpp>
#include <boost/numeric/itl/itl.hpp>


int main(int argc, char* argv[])
{
    using namespace mtl;

    mtl::par::environment      env(argc, argv);
    boost::mpi::communicator     world;

    typedef matrix::distributed<compressed2D<float> >   matrix_type;
    typedef mtl::vector::distributed<dense_vector<double> >  vector_type;

    matrix_type A;

    int n = 100;
    laplacian_setup(A,n,n);

    vector_type x(num_rows(A),1.0),b;

    // Sparse matrix vector multiplication
    b = A * x;

    // Compute the two norm
    double mbnorm = two_norm(b);
    printf(" b vector l2norm %10.2f\n",mbnorm);

    // reset x vector to be zero
    x= 0;

    // Solve for x in Ax=b using a Krylov solver, BiCGStabilized.
    // Use the ILU_0 preconditioner
    itl::pc::ilu_0<matrix_type>      P(A);
    itl::cyclic_iteration<double> iter(b, 500, 1.e-6, 0.0, 5);
    bicgstab_2(A, x, b, P, iter);

    // Print an element of x (should be one)
    printf(" x[1] = %g (should be one)\n",x(1));

    return 0;
}
```

FIGURE 10.9

A version of the serial MTL4 code from Fig. 10.7 for running on a distributed-memory supercomputer using MPI is shown here. The matrix and vector types in lines 14–15 have been changed to distributed, and the print output has been restricted to rank 0 (lines 29, 43). All other pieces of the example code remain the same as in the serial version.

number of MPI ranks is greater than the number of I/O ranks, some MPI processes will write to the same file. For instance, if the number of I/O ranks specified by the user is one, all data for each MPI rank will be written to a single file. When datasets are written to multiple files, metadata connecting each file is also written so that a visualization tool can read the separate files as if they were one file.

```
0001 #include <hdf5.h>
0002 #include <math.h>
0003
0004 // particle data structure
0005 typedef struct particle3D {
0006   double x, y, z;   // coordinates
0007 } particle_t;
0008
0009 #define PARTICLE_COUNT 10
0010
0011 int main(int argc, char **argv)
0012 {
0013   // declare and initialize particle data
0014   particle_t particles[PARTICLE_COUNT];
0015   for (int i = 0; i < PARTICLE_COUNT; i++) {
0016     double t = 0.5*i;
0017     particles[i].x = cos(t);
0018     particles[i].y = sin(t);
0019     particles[i].z = t;
0020   }
0021
0022   // create HDF5 type layout in memory
0023   int mtype = H5Tcreate(H5T_COMPOUND, sizeof(particle_t));
0024   H5Tinsert(mtype, "x coordinate", HOFFSET(particle_t, x), H5T_NATIVE_DOUBLE);
0025   H5Tinsert(mtype, "y coordinate", HOFFSET(particle_t, y), H5T_NATIVE_DOUBLE);
0026   H5Tinsert(mtype, "z coordinate", HOFFSET(particle_t, z), H5T_NATIVE_DOUBLE);
0027
0028   // create data space
0029   hsize_t dim = PARTICLE_COUNT;
0030   int space = H5Screate_simple(1, &dim, NULL);
0031
0032   // create new file with default properties
0033   int fd = H5Fcreate("particles.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
0034   // create data set
0035   int dset = H5Dcreate(fd, "particle data", mtype, space, H5P_DEFAULT,
H5P_DEFAULT, H5P_DEFAULT);
0036   // write the entire dataset and close the file
0037   H5Dwrite(dset, mtype, H5S_ALL, H5S_ALL, H5P_DEFAULT, particles);
0038   H5Fclose(fd);
0039 }
```

FIGURE 10.10

Example of use of the HDF5 library for output in the HDF5 format. This example outputs an array of particle information to a file called "particles.h5" and places this data in the dataset called "particle data". The HDF5 namespace resembles a file system directory, where HDF5 groups are analogous to directories and HDF5 datasets are analogous to files.

10.6 MESH DECOMPOSITION

One of the most important and widely used libraries for partitioning a finite element mesh across multiple processes is the METIS family of graph and hypergraph partitioning software, consisting of METIS [16] and its parallel MPI based counterpart called ParMETIS [17]. An example of mesh

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 352

```
HDF5 "particles.h5" {
GROUP "/" {
   DATASET "particle data" {
      DATATYPE  H5T_COMPOUND {
         H5T_IEEE_F64LE "x coordinate";
         H5T_IEEE_F64LE "y coordinate";
         H5T_IEEE_F64LE "z coordinate";
      }
      DATASPACE  SIMPLE { ( 15 ) / ( 15 ) }
      DATA {
      (0): {
            1,
            0,
            0
         },
      (1): {
            0.877583,
            0.479426,
            0.5
         },
```

FIGURE 10.11

Output from executing h5dump on the "particles.h5" output by the code in Fig. 10.10.

partitioning using the Trilinos library is shown in Fig. 10.13. These partitioning software tools are ubiquitous in simulations with unstructured meshes in order to decompose the mesh across multiple MPI processes.

10.7 VISUALIZATION

One of the most important libraries for HPC users is the Visualization Toolkit (VTK) [18]. It provides hundreds of visualization algorithms, enabling application developers to create their own visualization tools. It includes support for scalars, vectors, and tensors as used in contours, streamlines, and hyperstreamlines, respectively. VTK also supports distributed-memory parallel processing using MPI and multithreaded parallel processing for SMP architectures. An example of the VTK in use is the ParaView visualization tool, discussed in Chapter 12.

10.8 PARALLELIZATION

The most important parallelization library for distributed-memory architectures is the MPI library. There are multiple vendor and open-source implementations of MPI. A C++ friendly interface MPI is available via Boost.MPI [12]. For SMPs the most important parallelization libraries are OpenMP and Pthreads.

10.9 SIGNAL PROCESSING

Among libraries providing discrete Fourier transform capability, the FFTW ("fastest Fourier transform in the West") is one of the most widely used. It was developed at the Massachusetts Institute of

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 353

```
0001 #include <stdio.h>
0002 #include <stdlib.h>
0003 #include <assert.h>
0004 #include <math.h>
0005 #include <string.h>
0006 #include <mpi.h>
0007
0008 // Silo output headers
0009 #include <silo.h>
0010 #include <pmpio.h>
0011
0012 void DumpDomainToFile(DBfile *db, float *field, int myRank,int nx,int ny);
0013 void DumpMetaData(DBfile *db, PMPIO_baton_t *bat, char basename[], int
numRanks);
0014 void *Test_PMPIO_Create(const char *fname, const char *dname, void *udata);
0015 void *Test_PMPIO_Open(const char *fname, const char *dname, PMPIO_iomode_t
ioMode, void *udata);
0016 void Test_PMPIO_Close(void *file, void *udata);
0017
0018 int main(int argc,char *argv[])
0019 {
0020
0021   int numRanks, myRank;
0022   MPI_Init(&argc, &argv) ;
0023   MPI_Comm_size(MPI_COMM_WORLD, &numRanks) ;
0024   MPI_Comm_rank(MPI_COMM_WORLD, &myRank) ;
0025
0026   // The total number of files to write out
0027   int numfiles = 4;
0028   if ( numfiles > numRanks ) numfiles = numRanks;
0029
0030   // The local structured mesh size of each rank
0031   int nx = 50;
0032   int ny = 50;
0033
0034   // The data to write
0035   float *field;
0036   field = (float *) malloc(sizeof(float)*nx*ny);
0037
0038   // Specify some initial data
0039   for (int i=0;i<nx*ny;i++) {
0040     field[i] = myRank*3.14;
0041   }
0042
0043   // The silo library handler
0044   DBfile *db;
0045
0046   // the output filename
0047   char basename[32];
0048   sprintf(basename,"output_file.000.pdb");
0049
0050   // the subdirectory where the data is written
0051   char subdirName[32];
0052   sprintf(subdirName,"data_%d",myRank);
0053
```

FIGURE 10.12

Example of parallel I/O using the Silo library and the portable binary database as the low-level I/O library. Each MPI rank has its own unique 2D data that needs to be output. The number of files written by the code is decided by the user by changing the variable "numfiles" in line 27. Changing this variable can change the time it takes to write the output. The optimal performance will change depending on the file system in the supercomputer, but is generally somewhere between the two extremes of having each MPI process write its own file and having all MPI processes write to just one file. Regardless of the number of files written, however, visualization tools like VisIt (discussed in Chapter 12) can read and tie the separate output files together using the metadata added in line 84.

```
0054   if ( numRanks > 1 ) {
0055     // Set up baton passing
0056     // Three handler routines control the parallel creation, opening, and
closing of the files.
0057     // These are named here: Test_PMPIO_Create, Test_PMPIO_Open,
Test_PMPIO_Close
0058     // They are defined at the end.
0059     PMPIO_baton_t *bat = PMPIO_Init(numfiles,
PMPIO_WRITE,MPI_COMM_WORLD,[illegible],
0060                                 Test_PMPIO_Create,
0061                                 Test_PMPIO_Open,
0062                                 Test_PMPIO_Close,
0063                                 NULL);
0064
0065     // Determine the I/O rank
0066     int myiorank = PMPIO_GroupRank(bat,myRank);
0067
0068     char fileName[64];
0069
0070     // If I/O rank is 0, the filename is as specified
0071     // Otherwise, give the filename an integer suffix
0072     if (myiorank == 0) {
0073       strcpy(fileName, basename);
0074     } else {
0075       sprintf(fileName, "%s.%03d", basename, myiorank);
0076     }
0077
0078     // Wait for the turn to write data to the file
0079     db = (DBfile*)PMPIO_WaitForBaton(bat, fileName, subdirName);
0080
0081     DumpDomainToFile(db, field, myRank,nx,ny);
0082
0083     if (myRank == 0) {
0084       // Dump necessary metadata
0085       DumpMetaData(db, bat, basename, numRanks);
0086     }
0087
0088     // Finish writing, give someone else a turn to write
0089     PMPIO_HandOffBaton(bat, db);
0090
0091     PMPIO_Finish(bat);
0092   } else {
0093     // Only one rank in this case, no parallel I/O needed
0094     int             driver=DB_PDB;
0095     db = (DBfile*)DBCreate(basename, 0, DB_LOCAL,"test data", driver);
0096     if (db) {
0097        DumpDomainToFile(db, field, myRank,nx,ny);
0098        DBClose(db);
0099     }
0100   }
0101
0102   free(field);
0103
0104   MPI_Finalize();
0105   return 0;
```

FIGURE 10.12 Cont'd

```
0106
0107 }
0108
0109 void DumpDomainToFile(DBfile *db, float *field, int myRank,int nx,int ny)
0110 {
0111
0112   // allocate the coordinate arrays
0113   float *nodex,*nodey;
0114   nodex = (float *) malloc(nx*sizeof(float));
0115   nodey = (float *) malloc(ny*sizeof(float));
0116   int dimensions[2];
0117   dimensions[0] = nx;
0118   dimensions[1] = ny;
0119
0120   float *coordinates[2];
0121   const char *coordnames[2];
0122
0123   coordnames[0] = "x";
0124   coordnames[1] = "y";
0125
0126   // Give the local data some x and y coordinates
0127   for (int i=0;i<nx;i++) {
0128     nodex[i] = 1.0*(myRank*nx + i);
0129   }
0130   for (int i=0;i<ny;i++) {
0131     nodey[i] = 1.0*(myRank*ny + i);
0132   }
0133   coordinates[0] = nodex;
0134   coordinates[1] = nodey;
0135
0136   static char     meshname[] = {"mesh"};
0137   DBPutQuadmesh(db,meshname,coordnames,coordinates,
0138                 dimensions,2,DB_FLOAT,DB_COLLINEAR,NULL);
0139
0140   char fname[80];
0141   sprintf(fname,"testvar");
0142
0143   DBPutQuadvar1(db, fname, meshname,field,
0144          dimensions,2,NULL,0,DB_FLOAT,DB_NODECENT,NULL);
0145
0146   free(nodex);
0147   free(nodey);
0148
0149   return;
0150 }
0151
0152 void DumpMetaData(DBfile *db, PMPIO_baton_t *bat,
0153                          char basename[], int numRanks)
0154 {
0155
0156   // We only write out on variable in this example, called "testvar"
0157   int numvars = 1;
0158   char vars[numvars][32];
0159   sprintf(vars[0],"testvar");
```

FIGURE 10.12 Cont'd

```
// These objects provide the metadata needed to tie together
// data from multiple files
// the 'multi' objects tell us where the mesh and variables are written
// in the files directory
char **multi_mesh;
char ***multi_var;
multi_mesh = malloc(numRanks*sizeof(char*));
multi_var = malloc(numvars*sizeof(char**));
for(int v=0 ; v<numvars ; ++v) {
   multi_var[v] = malloc(numRanks*sizeof(char*));
}

// the 'type' objects tell us the type of mesh and variables written
int *typemesh;
int *typevar;
typemesh = malloc(numRanks*sizeof(int));
typevar = malloc(numRanks*sizeof(int));

// We start from the root directory in the silo file
DBSetDir(db, "/");

// Specify the type of mesh and variable being written
for(int i=0 ; i<numRanks ; ++i) {
  multi_mesh[i] = malloc(64*sizeof(char));
  typemesh[i] = DB_QUADMESH;
  typevar[i] = DB_QUADVAR;
}
for(int v=0 ; v<numvars ; ++v) {
  for(int i=0 ; i<numRanks ; ++i) {
    multi_var[v][i] = malloc(64*sizeof(char));
  }
}

// Indicate where in the file hierarchy to write the mesh and data
for(int i=0 ; i<numRanks ; ++i) {
  int iorank = PMPIO_GroupRank(bat, i);
  if (iorank == 0) {
    snprintf(multi_mesh[i], 64, "/data_%d/mesh", i);
    for(int v=0 ; v<numvars ; ++v) {
snprintf(multi_var[v][i], 64, "/data_%d/%s", i, vars[v]);
    }

  } else {
    snprintf(multi_mesh[i], 64, "%s_%03d:/data_%d/mesh",
             basename, iorank, i);
    for(int v=0 ; v<numvars ; ++v) {
       snprintf(multi_var[v][i], 64, "%s_%03d:/data_%d/%s",
                basename, iorank, i, vars[v]);
    }
  }
}

// write out the metadata
DBPutMultimesh(db, "mesh", numRanks, (const char**)multi_mesh, typemesh,
NULL);
```

FIGURE 10.12 Cont'd

```
0215
0216    for(int v=0 ; v<numvars ; ++v) {
0217       DBPutMultivar(db, vars[v], numRanks, (const char**)multi_var[v], typevar,
NULL);
0218    }
0219
0220    for(int v=0; v < numvars; ++v) {
0221      for(int i = 0; i < numRanks; i++) {
0222        free(multi_var[v][i]);
0223      }
0224      free(multi_var[v]);
0225    }
0226
0227    // Clean up
0228    for(int i=0 ; i<numRanks ; i++) {
0229      free(multi_mesh[i]);
0230    }
0231    free(multi_mesh);
0232    free(multi_var);
0233    free(typemesh);
0234    free(typevar);
0235
0236    return;
0237 }
0238
0239 void *Test_PMPIO_Create(const char *fname,
0240              const char *dname,
0241              void *udata)
0242 {
0243     // This is where the file is created.
0244     // We overwrite ("clobber") any existing files with the same name that
might
0245     // be in the way
0246     int            driver=DB_PDB;
0247     DBfile* db = DBCreate(fname, DB_CLOBBER, DB_LOCAL, NULL, driver);
0248
0249     // All data is placed in the dname subdirectory.
0250     if (db) {
0251       DBMkDir(db, dname);
0252       DBSetDir(db, dname);
0253     }
0254     return (void*)db;
0255 }
0256
0257 void *Test_PMPIO_Open(const char *fname,
0258             const char *dname,
0259             PMPIO_iomode_t ioMode,
0260             void *udata)
0261 {
0262    // This is where we open the file for appending to each.
0263    DBfile* db = DBOpen(fname, DB_UNKNOWN, DB_APPEND);
```

FIGURE 10.12 Cont'd

```
0264
0265     // All data is placed in the dname subdirectory.
0266     if (db) {
0267       DBMkDir(db, dname);
0268       DBSetDir(db, dname);
0269     }
0270     return (void*)db;
0271 }
0272
0273 void Test_PMPIO_Close(void *file, void *udata)
0274 {
0275   // Here the file is closed
0276   DBfile *db = (DBfile*)file;
0277   if (db)
0278     DBClose(db);
0279 }
```

FIGURE 10.12 Cont'd

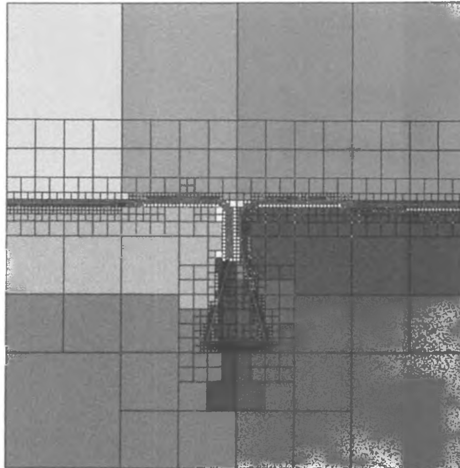


FIGURE 10.13

Example of partitioning algorithm using the Zoltan library. The different regions indicate the different partitions of the mesh domain.

Courtesy Lawrence C Musson at Sandia National Laboratories

Technology by Matteo Frigo and Steven Johnson, and provides discrete sine/cosine transform, discrete Fourier transform, and Hartley transform. It is optimized for speed by means of a special-purpose codelet generator called "genfft", which actually produces the C code that is used. FFTW supports SMP architectures with threads and distributed-memory architectures with MPI. It is used in two

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 359

```
0001 #include <fftw3-mpi.h>
0002 #include <stdlib.h>
0003 # include <stdio.h>
0004 #include <sys/stat.h>
0005 #include <fcntl.h>
0006 # include <time.h>
0007 #include <math.h>
0008
0009 int main(int argc, char **argv){
0010   const ptrdiff_t N0 = 8589934592; // 2^33
0011   fftw_plan plan;
0012   fftw_complex *data;
0013   ptrdiff_t alloc_local, local_n0, local_0_start,local_no,local_o_start, i, j;
0014   MPI_Init(&argc, &argv);
0015   fftw_mpi_init();
0016
0017   // This tells us the local size for each process
0018   alloc_local = fftw_mpi_local_size_1d(N0, MPI_COMM_WORLD,FFTW_FORWARD,
FFTW_ESTIMATE,
0019                      &local_n0, &local_0_start,&local_no,&local_o_start);
0020
0021   // Allocate the data
0022   data = (fftw_complex *) fftw_malloc(sizeof(fftw_complex) * alloc_local);
0023
0024   // This creates the plan for the forward FFT
0025   plan = fftw_mpi_plan_dft_1d(N0, data, data, MPI_COMM_WORLD, FFTW_FORWARD,
FFTW_ESTIMATE);
0026
0027   // Initialize the input complex data to some random numbers between 0 and 1
0028   for (i = 0; i < local_n0; ++i) {
0029     data[i][0]= rand() / (double)RAND_MAX;
0030     data[i][1]= rand() / (double)RAND_MAX;
0031   }
0032   // Compute an unnormalized forward FFT
0033   fftw_execute(plan);
0034
0035   // Clean up
0036   fftw_destroy_plan(plan);
0037   fftw_free(data);
0038   MPI_Finalize();
0039   return 0;
0040 }
```

FIGURE 10.14

Example parallel one-dimensional discrete Fourier transform using FFTW with MPI.

widely distributed molecular dynamics toolkits, NAMD [52] and Gromacs [53]. An example of a parallel one-dimensional complex discrete Fourier transform using FFTW is shown in Fig. 10.14.

10.10 PERFORMANCE MONITORING

The Performance API (PAPI) [20] provides tools for performance measurement and portable access to hardware performance counters for monitoring software performance. For many users the PAPI performance counters most frequently encountered are those which measure the L1 data cache misses

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 360

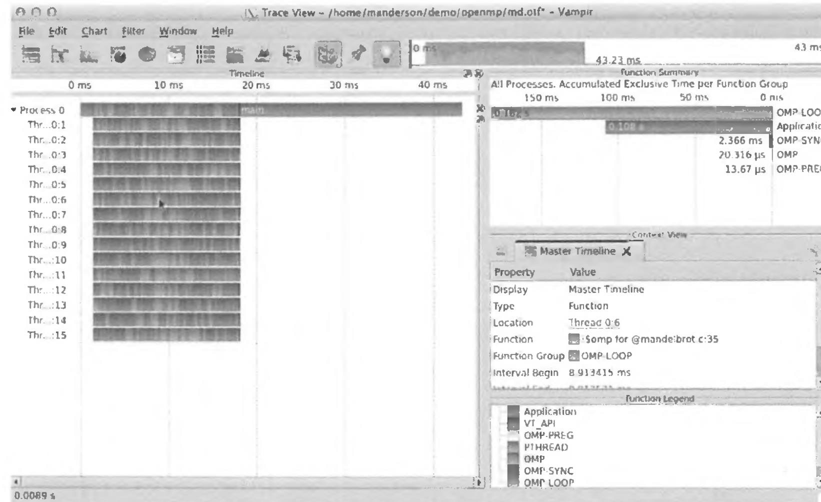


FIGURE 10.15

Performance-monitoring timeline of an OpenMP code run using 16 OpenMP threads within the VampirTrace framework. The timeline of execution for each thread is shown on the left, while the time summary of application functions and OpenMP looping is shown on the upper right. Context information for a region in the execution timeline can be examined, and is shown on the lower right above the function legend.

(*PAPI_L1_DCM*), the L2 data cache misses (*PAPI_L2_DCM*), and the number of floating-point operations executed (*PAPI_FP_OPS*). The PAPI library provides an important tool for users to diagnose performance issues via hardware counters from the bottom up in a portable way.

Other performance-monitoring tools like VampirTrace [21] can interface with PAPI as well as instrument MPI, OpenMP, and Compute Unified Device Architecture codes to provide a timeline of execution complete with messages and threads, illustrated in Fig. 10.15 using the Vampir performance-visualization tool.

10.11 SUMMARY AND OUTCOMES OF CHAPTER 10

- Several software libraries have been developed for HPC resources to fill specific computing needs, so application developers do not have to waste time redeveloping supercomputing software that has already been developed elsewhere.
- Apart from acting as a repository for software reuse, libraries serve the important role of providing a knowledge base for specific computational science domains.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 361

- Libraries become community standards and serve as ways for members of the community to communicate with one another.
- BLAS provides a standard interface to vector, matrix−vector, and matrix−matrix routines that have been optimized for various computer architectures.
- BLAS Level 1 involve vector operations. The naming scheme is a *cblas_* after which a precision prefix is placed before the operation name. Operations include dot products, norms, and rotations, among others.
- BLAS Levels 2 and 3 operations involve matrices and incorporate the type of matrix they support in their name. Levels 2 and 3 names take the form *cblas_*pmmoo, where the p indicates the precision, mm indicates the matrix type, and oo indicates the operation.
- Lapack incorporates BLAS Levels 2 and 3 to provide full problem drivers such as eigenvalue problems and linear solvers. A high performance version of Lapack is available: ScaLapack.
- Multiple additional widely used libraries exist which specifically target HPC resources. This chapter summarizes 25 such mature libraries to give a small sampling of what is currently available.

10.12 EXERCISES

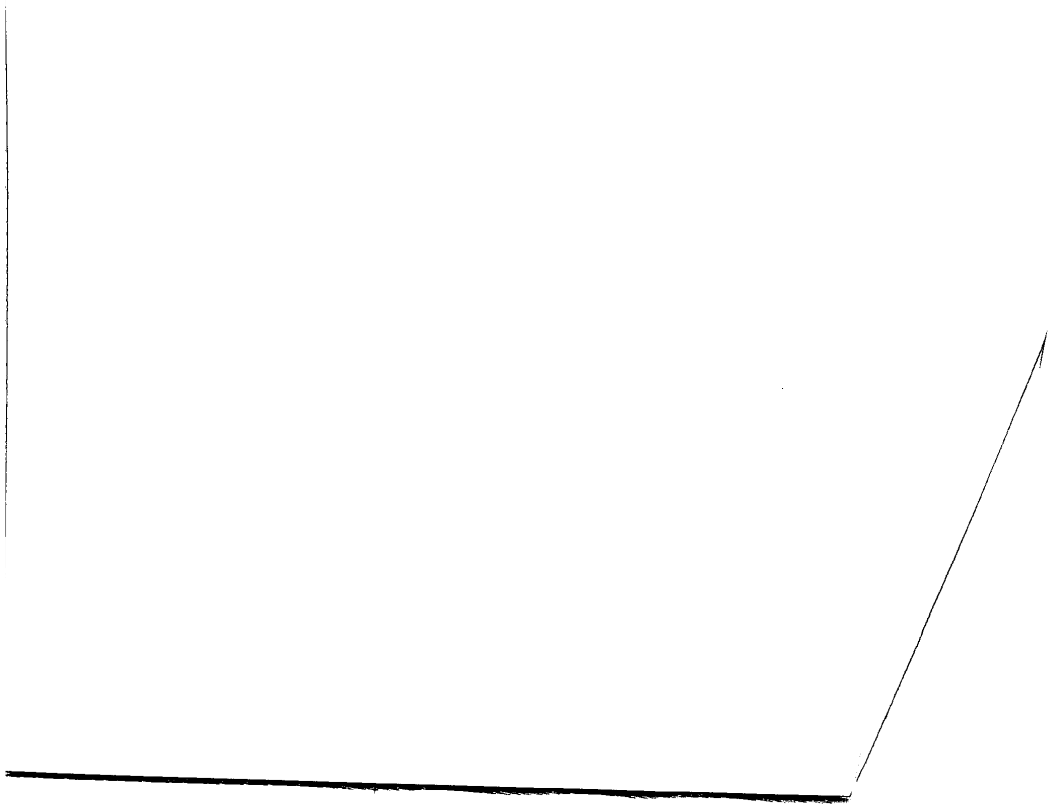
1. Explore the performance of matrix−matrix multiplication using the BLAS Level 3 dgemm routine for increasingly larger matrix sizes. Start with a randomly generated dense 3×3 matrix and incrementally increase the matrix size. For timing comparison, compute the matrix−matrix multiplication yourself just using for-loops without any BLAS calls for each matrix size explored. For each matrix size, which performs better, and by how much? Produce a plot comparing the time to solution for matrix−matrix multiplication with and without BLAS for each matrix size explored.
2. Using the DLATMR routine in Lapack to generate random square test matrices, compute the vector $b = Au$ where u is a vector whose elements are all 1 and A is the random matrix. Then use Lapack to solve the linear system $Ax = b$ for x. Check the solution to be sure that all elements of x are 1 after the solve. Produce a plot of the performance for solving $Ax = b$ for a wide variety of matrix sizes, beginning with 3×3 and exploring both symmetric and nonsymmetric test matrices.
3. Use PETSc and MPI to compute the sparse matrix vector product of a matrix and vector with randomly generated elements on a distributed-memory architecture. Select several sparse matrices to explore from the Matrix Market repository [54], and plot the time to solution as a function of the number of MPI processes used for the solve.
4. Write a code using the HDF5 library to read in the *particles.h5* file that is generated by running the code in Fig. 10.10.
5. Modify the FFTW code in Fig. 10.14 to compute the backward transformation, and then compare the input data prior to the forward transformation against the data that has gone through the forward and backward transformation.
6. Extend the Silo I/O example in Fig. 10.12 to support parallel 3D I/O. Test it by having each MPI rank write 3D data to file, instead of just 2D data as was done in Fig. 10.12.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 362

REFERENCES

[1] XSEDE, Extreme Science and Engineering Discovery Environment, [Online]. www.xsede.org.
[2] BLAS (Basic Linear Algebra Subprograms), [Online]. http://www.netlib.org/blas/.
[3] LAPACK — Linear Algebra PACKage, [Online]. http://www.netlib.org/lapack/.
[4] ScaLAPACK — Scalable Linear Algebra PACKage, [Online]. http://www.netlib.org/scalapack/.
[5] GSL – GNU Scientific Library, [Online]. https://www.gnu.org/software/gsl/.
[6] SuperLU developers, SuperLU, [Online]. http://crd-legacy.lbl.gov/~xiaoye/SuperLU/.
[7] PETSc Team, Portable, Extensible Toolkit for Scientific Computation, [Online]. https://www.mcs.anl.gov/petsc/.
[8] Universitat Politècnica de València, SLEPc, the Scalable Library for Eigenvalue Problem Computations, [Online]. http://slepc.upv.es/.
[9] Max Planck Computing and Data Facility, Eigenvalue SoLvers for Petaflop-Applications, [Online]. https://elpa.mpcdf.mpg.de/.
[10] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Hypre, [Online]. http://acts.nersc.gov/hypre/.
[11] The Trilinos Project, The Trilinos Project, [Online]. trilinos.org.
[12] Boost.org, Boost Home Page, [Online]. www.boost.org.
[13] The HDF5 Group, The HDF5 Home Page, [Online]. https://support.hdfgroup.org/HDF5/.
[14] NetCDF, Network Common Data Form Home page, [Online]. http://www.unidata.ucar.edu/software/netcdf/.
[15] Lawrence Livermore National Laboratory, Silo: A Mesh and Field I/O Library and Scientific Database, [Online]. https://wci.llnl.gov/simulation/computer-codes/silo.
[16] G. Karypis, METIS – Serial Graph Partitioning and Fill-reducing Matrix Ordering, [Online]. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.
[17] ParMETIS – Parallel Graph Partitioning and Fill-reducing Matrix Ordering, [Online]. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.
[18] The Visualization Toolkit, [Online]. http://www.vtk.org/.
[19] fftw.org, FFTW, [Online]. http://www.fftw.org/.
[20] The University of Tennessee, Performance Application Programming Interface, [Online]. http://icl.cs.utk.edu/papi/.
[21] GWT-TUD GmbH, VAMPIR, [Online]. www.vampir.eu.
[22] Sandia National Laboratories, Dakota – Algorithms for Design Exploration and Simulation Credibility, [Online]. https://dakota.sandia.gov/.
[23] Berkely Vision and Learning Center, Caffe – Deep learning framework, [Online]. http://caffe.berkeleyvision.org/.
[24] T.A. Davis, SuiteSparse: A Suite of Sparse Matrix Software, [Online]. http://faculty.cse.tamu.edu/davis/suitesparse.html.
[25] PISM Team, PISM: Parallel Ice Sheet Model, [Online]. http://www.pism-docs.org/wiki/doku.php.
[26] M. Adams, J. Brown, J. Shalf, B. Van Straalen, E. Strohmaier, S. Williams, High-Performance Geometric Multigrid, [Online]. https://hpgmg.org/fe/.
[27] R. Sampath, S. Adavani, H. Sundar, I. Lashuk, G. Biros, Dendro: Parallel Algoritms for Multigrid and AMR Methods on 2:1 Balanced Octrees, IEEE, Austin, Texas, 2008. SC.
[28] T. Kim, Cubica: A Toolkit for Subspace Deformations, [Online]. https://www.mat.ucsb.edu/~kim/cubica/.
[29] R. Schneider, FEINS: Finite Element Incompressible Navier-Stokes Solver, [Online]. http://www.feins.org.
[30] C. Curtin, Sanderson, Ryan, Armadillo: C++ Linear Algebra Library, [Online]. http://arma.sourceforge.net/.
[31] MUMPS: A MUltifrontal Massively Parallel Sparse Direct Solver, [Online]. http://mumps.enseeiht.fr/.

[32] C-SAFE and SCI, University of Utah, Uintah Software Suite, [Online]. http://uintah.utah.edu/.
[33] SimuNova, MTL4, [Online]. http://www.simunova.com/mtl4.
[34] K. Iglberger, et al., Blaze, [Online]. https://bitbucket.org/blaze-lib/blaze.
[35] Automatically Tuned Linear Algebra Software (ATLAS), [Online]. http://math-atlas.sourceforge.net/.
[36] J. Dongarra, [interv.] Thomas Haigh, April 26, 2004.
[37] netlib.org, LAPACKE, [Online]. http://www.netlib.org/lapack/lapacke.html.
[38] Lawrence Livermore National Laboratory, HYPRE, [Online]. http://computation.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods.
[39] SUNDIALS: SUite of Nonlinear and Differential/ALgebraic Equation Solvers, [Online]. http://computation.llnl.gov/projects/sundials.
[40] SAMRAI: Structured Adaptive Mesh Refinement Application Infrastructure, [Online]. http://computation.llnl.gov/projects/samrai.
[41] University of Oregon, TAU: Tuning and Analysis Utilities, [Online]. http://www.cs.uoregon.edu/research/tau/home.php.
[42] SC2000, Past Gordon Bell Award Prize Winners, [Online]. http://www.sc2000.org/bell/pastawrd.htm.
[43] Rice University, ARPACK software, [Online]. http://www.caam.rice.edu/software/ARPACK/.
[44] J. Dongarra, R.C. Whaley, Basic Linear Algebra Communication Subprograms, [Online]. http://www.netlib.org/blacs/.
[45] G. Kresse, et al., The Vienna Ab Initio Simulation Package, [Online]. https://www.vasp.at/.
[46] Mathworks, MATLAB, [Online]. https://www.mathworks.com/products/matlab.html.
[47] A. Azad, et al., Combinatorial BLAS, [Online]. http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/index.html.
[48] Apache, Apache Giraph, [Online]. http://giraph.apache.org/.
[49] Unidata, Network Common Data Form (NetCDF), [Online]. http://www.unidata.ucar.edu/software/netcdf/.
[50] The HDF Group, HDF5, [Online]. https://support.hdfgroup.org/HDF5/.
[51] S. Brown, PDBLib User's Manual, [Online]. https://wci.llnl.gov/codes/pact/pdb.html.
[52] Theoretical and Computational Biophysics Group, UIUC, NAMD: Scalable Molecular Dynamics, [Online]. http://www.ks.uiuc.edu/Research/namd/.
[53] Gromacs Project, Gromacs, [Online]. http://www.gromacs.org/.
[54] National Institute of Standards and Technology, The Matrix Market, [Online]. http://math.nist.gov/MatrixMarket/.



CHAPTER

OPERATING SYSTEMS

11

CHAPTER OUTLINE

11.1 Introduction **347**
11.2 Operating System Structures and Services **349**
11.2.1 System Components 349
11.2.2 Process Management 349
11.2.3 Memory Management 350
11.2.4 File Management 350
11.2.5 I/O System Management 351
11.2.6 Secondary Storage Management 351
11.3 Process Management **351**
11.3.1 Process States 352
11.3.2 Process Control Block 353
11.3.3 Process Management Activities 354
11.3.4 Scheduling 355
11.4 Threads **357**
11.5 Memory Management **358**
11.5.1 Virtual Memory 359
11.5.2 Virtual Page Addresses 359
11.5.3 Virtual Address Translation 359
11.6 Summary and Outcomes of Chapter 11 .... **361**
11.7 Exercises **362**

11.1 INTRODUCTION

A supercomputer is manifest visibly as a large room filled with many rows of many racks of many nodes of many cores, combined with the loud noise of myriad fans moving tons of air for cooling. But from the perspective of most users, who never actually see the physical high performance computing (HPC) system, the supercomputer is most readily viewed as the operating system (OS) and the user interface to it. In day-to-day usage patterns with a supercomputer, the OS gives the sense that it is the supercomputer itself. The OS owns the supercomputer.

An OS is a persistent program that controls the execution of application programs, as illustrated in Fig. 11.1. It is the primary interface between user applications and system hardware. The primary

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00011-3
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 366

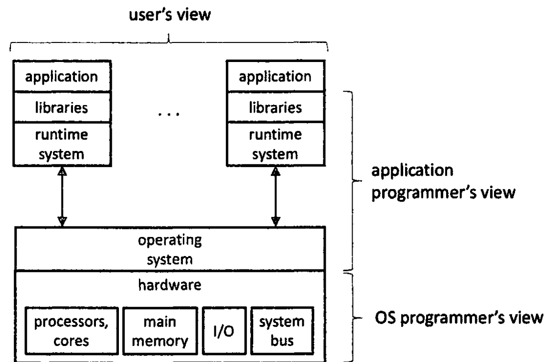


FIGURE 11.1

The primary functionality of the OS is to exploit the hardware resources of one or more processors, provide a set of services to system users, and manage secondary memory and I/O devices, including the file system.

functionality of the OS is to exploit the hardware resources of one or more processors, provide a set of services to system users, and manage secondary memory and input/output (I/O) devices, including the file system. The OS objectives are convenience for end users, efficiency of system resource utilization, reliability through protection between concurrent jobs, and extensibility for effective development, testing, and introduction of new system functions without interfering with ongoing service. The OS is one of the key layers in the total computer system stack, as illustrated in Fig.11.1.

Resources managed by the OS are the processors and their integrated cores, the main memory of the systems out of which the cores work, I/O modules, and the system bus. Processors within the same system may be of different classes, such as conventional processors (e.g., Xeon), lightweight cores (e.g., PHI), and graphics processing unit accelerators. Managing main memory is challenging, as it involves both the memory hierarchy and the virtual address space. Main memory usually refers to the banks of dynamic random access memory (DRAM) directly connected to the processors, but it also includes data movement between the main memory and the intervening cache hierarchy (two to four layers) and in the other direction to secondary storage. Memory objects are virtually addressed and the translation between virtual and physical addresses is the responsibility of the OS, including their placement in main memory and secondary storage. The I/O modules have diverse subsystems, including secondary storage and the file system hierarchy, communications equipment such as wide area networks, and terminals for user interactive control. The system bus provides communication between the processors, memory, and I/O.

The OS incorporates all services and facilities of a supercomputer.

- It holds the local directory and files system.
- It controls the allocation of the hardware resources.

- It governs the scheduling of user jobs.
- It stores temporary results.
- It provides many of the high-level programming tools and functions.
- It exports the user interface to the system for all user commands.
- It protects user programs from errors caused by other running applications.
- It supports user access to system I/O, networks, and remote sites.
- It provides firewalls for security of operation and data storage.

The OS performs all of these purposes and more to make a convenient, reliable system which delivers efficient and scalable performance. OS structures and interfaces can be complex, large, and complicated to use or program. The purpose of this chapter is to highlight specific capabilities, structures, and functionality that relate to effective HPC operation and use. It does not give exhaustive coverage of the entire OS, as the user is unlikely to encounter most of these aspects. An appendix, "The Essential Linux", describes the interface syntax that the user is likely to need in performing the hands-on examples and exercises. This chapter presents key aspects to understand the operation and use of the HPC OS, including:

- OS structures and services
- process management
- parallel threads
- memory management
- modern OS, Unix, and Linux.

11.2 OPERATING SYSTEM STRUCTURES AND SERVICES

OSs can be described in a number of ways. One is to represent the OS components and their interconnections, describing the data and control flow. Another way is to describe the services that are performed by the components comprising the OS. A third approach is to define the interface semantic constructs employed by users and programmers. This section introduces these ideas of OS structures and the services they provide.

11.2.1 SYSTEM COMPONENTS

OSs are complex software packages consisting of many separate but interrelated components. These components individually or in combination achieve the functions and deliver the services required by the users directly or for system management and control. While the OS may differ from machine to machine with low-level variation of means and methods, essentially all mainstream HPC OSs share the same major components. The following are representative of what one is likely to find in any of these computers.

11.2.2 PROCESS MANAGEMENT

User and system programs that are executing are made up of instances called "processes", which are instantiations of program procedures (code text). Many processes may operate concurrently under a

single OS, so the OS incorporates a major component responsible for managing them. The process management component controls the full lifecycle of a process running on the system hardware. It creates and ultimately terminates all the processes, whether provided by the end user or part of the functioning system itself. Throughout the lifecycle of a process, this component manages its scheduling of operation and allocation of processor resources, suspending and resuming processes as required to optimize a selected objective function for system operation. Processes can communicate among themselves, with the output results of one conveyed to the input of another. The process management component enables the paths of communication between processes (e.g., sockets). The control flow between processes requires process synchronization and variables supervised by this component. In addition, the processor resource allocation to processes is handled by the process management component.

11.2.3 MEMORY MANAGEMENT

By some measures, including cost, an HPC system is mostly data storage. Program data must reside within the memory system, which seen from the architecture perspective is a multilevel hierarchy including registers, buffers, and three layers of cache: main memory, which may be distributed among all nodes, secondary storage, which is still primarily hard disks but increasingly includes nonvolatile semiconductor storage technology, and tertiary storage employing tape cassettes and drives for archival storage. The tradeoffs of all these layers are speed of access and cost of capacity, with reliability and energy also being important. The OS is responsible for data allocation to memory resources and migration between levels. Memory management is also responsible for address translation between the virtual address blocks of program data, called pages, and blocks of physical storage, called frames. The OS manages the page table that maps the page numbers to the frame numbers. In case that a particular page is not in memory, that is a page fault occurs, the OS has to swap the frame from secondary storage into main memory and update the page table accordingly prior to the related process continuing.

11.2.4 FILE MANAGEMENT

The OS is responsible for users' data and programs organized in files through a hierarchy of named directories. The file system managed by the OS presents this abstraction of the system to the user, and includes many more functions and used services. The system supports nonvolatile storage; that is, the information does not go away when the associated processes terminate. Ordinarily the file system resides on secondary storage, which is primarily hosted by hard-disk drives. However, newer systems may include nonvolatile random access memory (NVRAM) semiconductor devices for lower power consumption and faster response, sometimes dedicated to metadata for large graph structures. In the case of laptops, pads, and phones, solid-state devices made from these components may constitute all the file system. File management may also involve tertiary storage in the form of tape robots. The cost per byte of such storage is much lower than the cost for other forms, with much higher density, and it is therefore are perfect for archival storage of files. Initial access times can be measured in minutes, however, so the OS supports the user's file management system across a complex multilevel storage system, and possibly mounts external file systems as well for even greater data storage space.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 369

11.2.5 I/O SYSTEM MANAGEMENT

The OS is responsible for managing all sources and destinations of data flow in and out of the computer it supports. The file system is just one example of the I/O system employing secondary storage. Users are most familiar with the standard I/O that gives them direct interactive access with the system by a minimalist command-line interface or the increasingly common windows-based graphical interfaces. Web browsers access the internet through additional I/O devices (e.g., Ethernet) from which much of the external data is acquired, also supported by the OS I/O management. For clusters and massively parallel processors, at the lowest level the system area network (e.g., Infiniband) for each node is the I/O channel that connects it to all the other system nodes comprising the total supercomputer and again managed by the OS I/O system. Many other devices are also supported, as described in the Chapter 6 on architecture. Some of these are for maintenance and are not visible to the users; others are as simple as switches and lights.

11.2.6 SECONDARY STORAGE MANAGEMENT

As mentioned, the OS is responsible for secondary storage. Usually comprising many hard-disk drives, but possibly also some solid-state NVRAM, secondary storage delivers high density and nonvolatility for long-term storage. The OS may manage access to local disks for each node or a separate part of the system of disks connected by a storage area network such as a redundant array of independent disks configuration (there are several) for higher access bandwidth and greater reliability through redundancy of storage. While secondary storage is important to users in its OS support for file systems, it also provides other services. Virtual memory, in which pages of data for a process may be temporarily stored in secondary storage, gives the impression of larger memory capacity, although the data pages are actually distributed between physical main memory and secondary storage. The OS also uses secondary storage to buffer processes for future scheduling, or sometimes when swapping jobs in and out of memory systems. In all these cases and more, the OS is responsible for managing secondary storage, providing interfaces to it, and including services.

11.3 PROCESS MANAGEMENT

A process is a program or subprogram in execution. It is a unit of work within the system that is performed to completion. A program is a passive entity; principally a block or blocks of binary code produced by a compiler from a high-level representation of an application to the low-level machine-interpretable form for machine execution. A process is the instantiation of a program within a computer as an active entity of work in progress. It consumes resources and combines program blocks with data representing both its current operating state and the information upon which the program is to operate. The OS is responsible for process management: where the data elements of the process are, its current control state and intermediate values, and how the process is related to both its parent (calling) process and possibly its child processes (those which it has called). This section introduces the elements and mechanisms of OS process management.

A process needs resources to accomplish its task(s): both hardware functionality and logical objects defining the state and direction of the process. A process needs to have allocated to it such hardware as one or more central processing units (CPUs), memory for both data being processed and program

blocks, access to I/O ports and devices, and files in mass storage where input data to start the program, output data to store the process results, and possibly additional storage for intermediate results are handled.

The logical resources that ultimately specify the process include a number of data objects. The program counter points to the next instruction to be executed (an address) within the program code block. The code section itself describes the operations of the process to be performed. The process stack contains localized data of direct importance to the process specification and intermediate values, including such things as return addresses upon completion. The data section contains the global variables of the user computation. When the process terminates, all the reusable resources allocated to it are reclaimed by the OS for use by future processes.

Section 11.4 introduces the idea of a thread, which is itself an executable within the context of a process. If the process has only one thread, it includes only a single program counter defining the location of the next instruction to be performed. It is possible for a process to have more than one active thread, in which case the process contains multiple program counters; one counter for each operating thread.

A modern system, whether an enterprise server or a basic laptop, has a number of concurrent processes operating at the same time. Some are user processes, possibly in support of applications with multiple users; others are system processes providing services directly in response to user application requests or to support the management of system resources. Curiously, one process is responsible for the management of all these processes. These OS process management tasks are discussed in the following subsections.

11.3.1 PROCESS STATES

At any one time each activated process managed by the OS exists in one of a number of states, depending on its current condition and activity. These process states are mutually exclusive and collectively exhaustive, in that they fully describe the possible lifecycles of a given process. Different OSs are distinguished in part by the possible process states each supports and employs in guiding the evolution of its constitutive processes, but they exhibit many similarities. Here a relatively simple machine in a fully functional state is considered to illustrate OS-supported process states, as shown in the diagram below. All OSs will include these states or multiple states. For example, the Linux OS presented in Appendix B has a more diverse state structure, but all the states in this diagram can be mapped on top of the Linux state diagram.

When a new process is initiated for the first time for a specified program, it enters into the *new* state among the process states. In this state the process is being created and the necessary memory objects fully designing the process are being allocated and populated. When this has been accomplished, the process transitions into the *ready* state. In a symmetric manner, when the process has completed all work associated with it and deposited its results in the appropriate locations for future use, it enters the *terminated* state. Once in this state, the process is known to have finished execution. At this point the OS modifies its control tables to eliminate the context of the process and reclaim the physical and logical resources associated with the process.

The *running* state of the process is that condition under which the process is actually executing its instructions on the data associated with it. When running, the process is making progress toward completion of its workload. If in this state it reaches the point of completion, it transitions to the

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 371

terminated state as described above. However, it is possible that other events will occur and require the process to suspend temporarily and resume at a later time. One of these circumstances can be an asynchronous external interrupt. An interrupt is a signal from any of several sources indicating that another process has immediate priority, such as an OS service routine that must be engaged for the system as a whole to progress. An interrupt will cause the current process in the running state to exit the processing resources (e.g., CPU) and transit to the *ready* state. Alternatively, if a process in the running state experiences a need to delay because of a wait event or an I/O request that may take tens of milliseconds, then if the process remained in the running state it would waste precious computing resources due to the delay caused by these conditions. Instead, the process will transition from the running state to its *waiting* state.

The waiting state is that condition of the process assumed when it is unable to proceed immediately with its execution because of a delay of a pending service, access such as I/O requests to mass storage, or a need for user input. Once entered, a process remains in the waiting state until the source of the delay is cleared by some external action (e.g., the arrival of the data requested from secondary storage). In this way other processes can enter the running state and take advantage of the processor resources for greater efficiency of system usage. When the delaying condition is satisfied and the process can proceed forward, it is unlikely that the computing resources are immediately available as one or more other processes are likely to be actively using them. Thus the process that had been in the waiting state transitions to the *ready* state of the process lifecycle. The OS draws upon the processes in the ready state to select the next process to be placed in the running state. Many processes may be pending in the running state, waiting for their turn to begin executing either for the first time after originating from the new state or resuming, having previously been in the running state at some time in the past. It is typical for a process to cycle back and forth among the three states, ready, running, and waiting, prior to completing its workload and finally entering the terminated state. In this way the user gets the impression that any number of processes are computing concurrently, when in fact they are time-sharing the physical resources but switching states so quickly that they all appear to be making progress towards their end computational goals.

11.3.2 PROCESS CONTROL BLOCK

Each process being managed by an OS is represented by a dedicated data structure referred to as a "process control block" (PCB). Like the process state machine, the PCB will vary from OS to OS. However, there are a minimum number of basic parameters common to the PCBs of all OSs. The PCB contains the data that specifies the existence of a particular process and the information necessary to permit the process to make forward progress.

From the previous subsection, it is clear that the process state is a critical parameter determining the modality of a process at any point in time and thus the possible states to which it may transit. The PCB contains a field that specifies an encoding of all possible process states, and holds the value associated with the state of the process as the process evolves throughout its lifecycle.

Calling parent process pointer provides the link to the active process that was responsible for the instantiation of the current process represented by the PCB. This pointer link is either the name of the parent or calling process (*process number*) or the virtual address of the PCB of the parent process. The process number is an arbitrary positive integer that is unique among all processes running in the system at any one time. The process pointers combined with the PCBs of an entire program form

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 372

a tree describing the operating state of the program (user or system), with the PCBs as the vertices (nodes) of the tree and the pointers as the links. The next instruction to be executed by the process is represented in the PCB by the *program counter.* This can either be the virtual address of the next instruction or a combination of the virtual address of the program code block and the offset within the code block of the next instruction. This is updated, often incremented, after every instruction issue.

Registers are the highest level of the memory hierarchy (or lowest, depending on how you draw it) and hold the most important values of a process execution at any one time. Registers have their own namespace (register number) and update their value contents through load and store instructions. When a process is suspended (to either waiting or ready state), the values of the physical registers must be copied to the PCB, from which the registers can be restored when the process restarts in the running state. The other main data of the process is stored in main memory. It may include a stack frame assigned to the process and the blocks of main memory that it uses and possibly shares with other processes. Because the process data is in main memory, it does not need to be copied in the PCB; but the locations of such data may be required, including pointers to the head of the associated data blocks and the limits of those blocks and the stack frame.

Other information that is usually incorporated in a PCB specification includes accounting details associated with the program system resource usage, such as CPU utilization, memory capacity employed, secondary file storage space, priority, user information, and other characterizing data. In addition, the PCB holds information about the process related to the I/O devices allocated to it, including a list of all the open files it is currently accessing. With the PCB, a process can be restarted from any of its passive states to the running state at any time.

11.3.3 PROCESS MANAGEMENT ACTIVITIES

The OS is responsible for a number of services associated with all the processes active in the system. Implicit in these services is the simple overarching task of keeping track of all the active processes on the system and the resources of the physical system (e.g., CPUs, memory blocks, files, etc.) that are allocated to the processes. Foremost among these management activities is the creation and termination of processes—both user and system types. The creation of a process, that is its instantiation, is in response to a call either by the system (including user command-line requests) or the user program giving initial data. A new PCB is produced by the OS for instantiation of the specific process, with fields filled as previously discussed. Other resources such as additional memory blocks, file accesses, and I/O sockets may also be allocated. The termination of a process by the OS involves the deletion of the PCB and other assigned resources to free them up for future computations.

To achieve higher efficiency by making better use of the physical resources of the computer system, the OS supports process context switching. This requires suspending and resuming a process. A process can be suspended for a number of reasons. One is multiprogramming, where one or a few computing resources are shared by a much larger number of active processes so all of them are operating concurrently, making progress toward their conclusion in sufficiently small time steps such that the user experiences a sense of continued operation, but the time steps are large enough to avoid the deleterious effects of the overhead time associated with the action of context switching. Another important factor motivating context switching is the avoidance of resource blocking in the presence of operations imposing extended delays. Such waiting times can greatly degrade the efficiency of system usage. This is predominantly associated with I/O tasks, such as reading a file from secondary storage or

waiting for real-time user input from a standard I/O. Operations like this can take hundreds of milliseconds or multiple seconds depending on the specific nature of the requests and contention for shared resources by other processes. When a process is so engaged, the OS relinquishes its dedicated computing resources (e.g., processor core), putting the process state in memory while simultaneously allocating those same resources to another pending process waiting for access to make progress. Upon completion of the delaying service request of the original process, the OS resumes activation of the process in its turn, based on scheduling mechanisms and policies (discussed below).

Processes often work together, cooperating on a single program and sometimes sharing mutable data and other resources. To do so correctly they must occasionally synchronize, so computational work is done in the right order. For example, if two processes share information in memory, they must coordinate to ensure that one does not read data until the other has written it (read-before-write hazard), or conversely that one does not write to a memory location until the other process has accessed it (write-before-read hazard). The OS supports synchronization mechanisms such as barriers, semaphores, and mutexes, among others, by which two or more processes may order their respective computations to avoid these possible hazards.

Processes may directly communicate with each other, passing or exchanging messages or data streams. Mechanisms for conducting such message passing between concurrent processes are provided by the OS, including the user interface calls that give control to the user program. Sockets are one example of the class of constructs used for this purpose, and are found in many but not all modern OSs. This is one case where multiprogramming becomes critical. If one process is active (occupies the processor) but requires a message from another process that is suspended, without appropriate OS control a deadlock condition could occur, precluding forward progress.

11.3.4 SCHEDULING

At the heart of the process management services supported by the OS and described in earlier sections is the cross-cutting functionality of process scheduling: the determination of what processes are given the necessary physical resources to run and when they are allocated. With a number of processes active (running, pending, or suspended) at any one time and fewer executing resources than processes, this selection and control function is actually very difficult to perfect and has been the subject of countless research and engineering undertakings over many decades.

The job queue holds all processes, whatever their states, and any new process entering the system is put in the job queue by the OS. Only upon termination is a process eliminated from the job queue. At any one time a process may be ready for execution or waiting for an I/O device service call. The job queue consists of a number of subqueues (this varies somewhat among different OSs) in which processes may be temporarily held until specific requirements are satisfied. A representative structure of the job queue may include the following.

- Ready queue—holds pointer to PCBs of processes pending execution.
- Child queue—holds processes (PCB pointers) waiting for their respective child processes to terminate.
- Interrupt queue—includes processes waiting for interrupts to occur.
- Multiprogramming queue—processes that have used up their last timeslice and must delay a minimum amount of time prior to resuming execution.
- I/O queues—a queue for each I/O device holding PCBs of processes requiring that device.

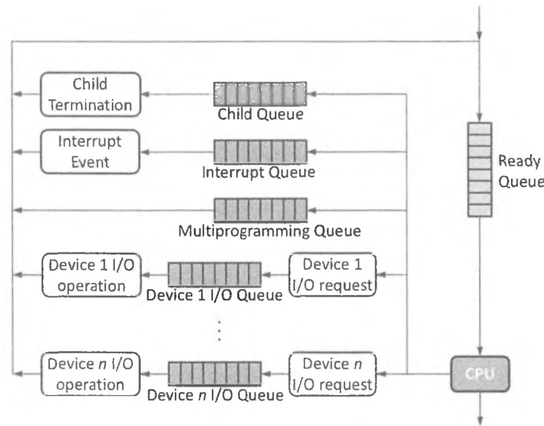


FIGURE 11.2

The job queue holds all processes, whatever their states, and any new process entering the system is put in the job queue by the OS. A representative job queue may include a ready queue, child queue, interrupt queue, multiprogramming queue, and I/O queues.

These are illustrated in Fig. 11.2. The OS is responsible for managing these (and potentially other) subqueues as part of the job queue structure. Each queue usually has two pointers, one to the head of the queue and one to its tail, if a first-in, first-out policy is employed, but other queue organizations are possible, such as a stack (last in, first out). The OS transfers the pointer to a process PCB from one queue to another as its condition state is altered.

The process scheduler incorporates a job scheduler where a job may comprise multiple processes. When many more jobs and their component processes exist than there are execution resources (e.g., processor cores), with total memory requirements that exceed the capacity of the physical memory (a common situation for a typical server), then many or perhaps a majority of the jobs must be spooled to secondary storage (typically hard disks) and only individually migrated to main memory when ready to be executed.

Typically, three job schedulers are employed to manage scheduling within the context of this memory hierarchy.

- Long-term scheduler—identifies processes or jobs from the spooler in mass storage to swap them into main memory in preparation for execution. Responsible for maintaining a balanced set of jobs, some I/O bound and some compute bound, to maintain even flow in all queues.
- Short-term scheduler—chooses the next process to be allocated computing resources for execution from those residing in the ready queue.
- Medium-term scheduler—swaps jobs or processes from main memory back into the mass-storage spooler when the priority of job ordering demands that main memory be freed up so new jobs can be included in the execution stream. Thus jobs may have to be swapped back on to the spooler on occasion.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 375

As previously discussed, context switching is an OS function that moves a process state in and out of the execution resources. Specifically, when the short-term scheduler is preparing to execute a process that is in the ready queue, the OS must first copy the state of the prior process that had been using the resources intended for the new process into the first process' PCB. Then the context of the new process in its PCB is loaded into the execution resources, such as processor registers.

11.4 THREADS

Threads present another level of parallelism control within a process. While processes are said to provide coarse-grain parallelism, threads provide the means of realizing medium-grain parallelism, which may give more parallelism, better scalability, and possibly shorter time to solution. Sometimes threads are referred to as "lightweight processes", but this text avoids this, as there are specific distinctions between threads and processes that distinguish their usage and effects. Finally, processes and threads map very nicely on to modern architectures using many nodes, each with a number of processor cores. A process can occupy a full node of cores with threads running on individual cores.

The thread state is reminiscent of that of a process included in the PCB, but is generally less complex. The thread state will include:

- a designator of the process of which it is a part
- a program counter indicating the next instruction to be executed by the thread
- a stack pointer to the frame of the pages directly related to the thread
- register contents.

This context data must be swapped in and out of the hardware resources as one thread is replaced by another in the executing processor cores.

There are two general categories of threads. Threads directly managed by the OS, as discussed above, are known as "kernel threads". The OS allocates specific kernel threads to underlying hardware processor cores. Because the OS is involved in the direct manipulation of the kernel thread, the overhead of its management is significant, even if less than the management of processes. The other kind is the "user thread", sometimes referred to as "runtime threads". These are not directly managed by the OS but rather by a runtime software system in user space. The overheads are smaller and the runtime system knows more about what the user job wants to do and how to do it better. The relationship between the two kinds of threads is that the runtime system allocates user threads to individual kernel threads. Usually (but not always) there is one kernel thread instantiated by the OS for each processor core. Except in the case of interrupts or multiprogramming, the kernel thread stays relatively static in this mapping, so few context switches of kernel threads are required. The runtime system allocates the kernel threads made available to it to the user threads for which it is responsible. There are several ways that it can do this, illustrated in Fig. 11.3. For example, all user threads may be assigned to a single kernel thread, sharing it one at a time. This is known as "all to one". In this case no parallelism is exploited within the application, but rather the kernel threads are running different jobs for job parallelism. A second case is when the runtime system may simply assign one user thread to each of multiple available kernel threads. In this case overheads are low and parallelism is exploited but no dynamic control is used, as might be necessary with irregular user threads. This is known as "one to one". The most general form of mapping of user threads to kernel threads is when the set of user threads is dynamically and possibly adaptively assigned to kernel threads as the workload requires and kernel threads become available. The policy guiding such a "many-to-many" strategy can become quite complex and is a subject of continued study.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 376

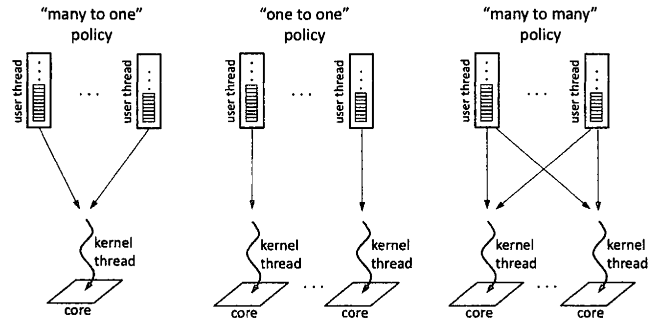


FIGURE 11.3

The runtime system allocates the kernel threads made available to it to the user threads for which it is responsible in several different ways.

11.5 MEMORY MANAGEMENT

While it may appear from the preceding text that the most important responsibility of the OS is management of execution resources and scheduling of jobs, processes, and threads on them, it can be argued that the management of the memory hierarchy is equally important and if anything more complex and demanding of OS functionality. Part of this is due to the disparity between the clock rates of the processor cores (complementary metal-oxide semiconductor technology) and the main memory (DRAM technology), which is between one and two orders of magnitude. With the separation of processor logic and main memory banks (the von Neumann bottleneck), full time to completion of memory access by cores can be between 100 and 200 processor core cycles. As discussed in Chapters 2 and 6 on architecture, modern computers and HPC systems incorporate multiple levels of storage to mitigate the latencies and bandwidth constraints imposed by this structure, and these are managed in part by the OS.

Memory management controls what data is in the main memory and registers at any particular time. Such data is usually stored in secondary storage organized in directories, files, and special blocks employed by the virtual memory page backing store. The OS ensures that all data needed for a scheduled process is in the main memory, as well as instructions for the executing code. The OS memory management subsystem maintains the tables (reference tables) that map the virtually addressed pages (discussed later) to the physical pages of memory and provide means for virtual address (to physical) translation. Memory management is an important part of achieving high utilization of the processor cores by maintaining the right data most likely to be accessed in the main memory. To achieve these goals, the OS memory management supports a multiplicity of activities, including control of allocation of physical memory to instantiated processes and deciding which memory pages should be swapped between memory and secondary storage to assign and reacquire memory space. Throughout these activities the OS must maintain its ability to support virtual address translation. This section expands on some of these key points.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 377

11.5.1 VIRTUAL MEMORY

Virtual memory is a powerful abstraction for naming memory blocks independent of their physical location, supported by the OS. It is implemented by the memory management part of the OS. Virtual memory controls the relationship and mapping of the logical (virtual) address of a page of data to the location of physical data storage, which can be either main memory or secondary storage (e.g., hard disks). The implementation of virtual memory over the history of computing has yielded several important advantages over direct user control of physical memory. Originally, with the amount of main memory being relatively small, the use of mass storage to give the appearance of a larger storage space than just the physical memory greatly simplified the programmer's task while allowing portability of code across systems of different scale, type, and generation. Over time and the use of multiprogramming and multitasking where a multitude of processes and jobs would be being performed concurrently, virtual memory systems provided protection of memory usage by different processes, giving each process its own virtual address structure and making sure that one process did not interfere with the data in another process. Virtual memory also allowed the OS to overlap the sequence of different processes, so while one process is executing another process memory content can be read into memory and possibly a previous process results are read at the same time, thus minimizing the lag between the execution of successive jobs.

11.5.2 VIRTUAL PAGE ADDRESSES

There are a number of ways to organize data for use and storage. Among these is the simple concept of a "page", which is a fixed-length contiguous block of data that can be mapped on to an equivalent-sized block of physical memory or similar space on secondary storage. Paging allows a process to consist of a collection of fixed-size blocks. Each page has a virtual address. Every unit of data (i.e., bytes, words) within a page is identified by the virtual address of the page, and by an offset index from the starting location of the page to the position of the data within the page. The virtual address of the page is simply a page number. The virtual address of the page, once assigned by the OS, remains a constant independent of whether the virtually addressed page is stored in main memory or secondary storage. The medium-term scheduler may cause a page and other pages related to a process to swap in and out of secondary storage from main memory. A virtual page may reside in different physical memory pages throughout the computation, as determined by the OS memory management function.

11.5.3 VIRTUAL ADDRESS TRANSLATION

The OS incorporates a *page table* which is central to the method for associating virtual page addresses (or page numbers) to physical locations within the main memory or secondary storage. The page table has one entry per page. This entry is determined by the assigned page number as an offset within the page table (the value of the offset cannot exceed the length of the page). The page table entry for a particular virtual page stores the memory frame number of a page frame (or physical page) within the main memory. A request to memory for data can always be located through the page table. When the virtual page migrates in physical space, its respective entry in the page table is updated by the OS.

While this works logically and is a necessary component of OS memory management, it alone is insufficient to deliver performance. Every access request using the page table requires an OS

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 378

function call, which is quite time consuming. For each instruction issue, which is about every processor core cycle, there must be a load of the instruction from the memory system. In addition, operational data needs to be loaded in the order of a quarter of the time (this varies per application workload). So the page table alone is inadequate to provide performance-oriented memory access in a virtually paged system.

The translation lookaside buffer (TLB), illustrated in Fig. 11.4, is an architecture means to deliver much of this needed performance, at least under favorable conditions. The TLB is a special-purpose cache that provides high-speed mapping of virtual page numbers to main memory frame numbers for recently used stored data. This in turn delivers very fast data access for memory loads and stores. Like a cache, the TLB exploits temporal locality where the physical addresses of recently used virtual pages are stored. The TLB has an associative access hardware that enables fast address translation and thus data access. Of course, regular data and instruction L1 caches provide most data access requests in one or two cycles as well, so on a good day (microcycle) virtually addressed data can be loaded in one or two cycles. Only when there is a TLB miss, when a particular virtual page number is not found in the TLB, does a page table access take place, with its significant overhead. In this case the OS accesses the page table, determines the correct frame number, and updates the TLB, at which point the memory access can continue as usual.

It is possible when locating a virtual page as a physical frame in main memory that there is no matching memory frame associated with the desired virtual page. This is referred to as a "page fault". It is a function of the OS to attempt to minimize page faults, as the cost of encountering one can be

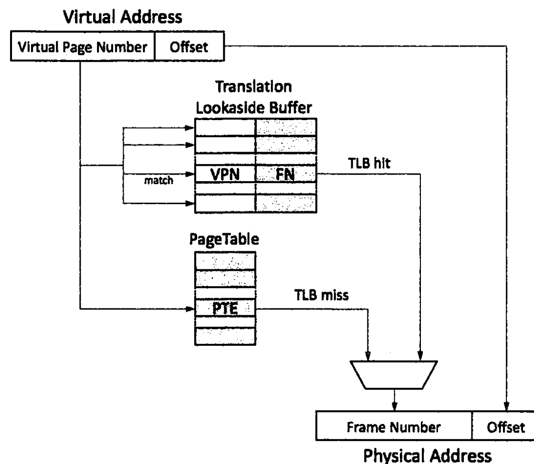


FIGURE 11.4

The translation lookaside buffer (TLB) is a special-purpose cache that operates to provide high-speed mapping of virtual page numbers to main memory frame numbers for recently used stored data.

significant. The combination of OS overheads and the transfer of a frame from secondary storage to main memory can take tens to hundreds of milliseconds. The OS bringing frames in from secondary storage when needed is known as *demand paging*. OS policies are applied to minimize the page faults. But programmers can help the system operate more efficiently in this regard by constructing workflows such that they take best advantage of data reuse and thereby limit the TLB misses, cache misses, and page faults.

Virtual-memory OSs provide a seamless way to take advantage of secondary storage without explicit user intervention through process and page swapping. In terms of user productivity this can be a very good thing; in terms of system performance it can be a very bad thing. The OS provides a powerful functionality in the form of a directory and file system that enable users to store and manage their data in a nonvolatile form. Indeed, a user's perspective of a supercomputer is largely enabled by the file system through the user interface, whether command line or point-and-click windows. The OS file system is so important to HPC that Chapter 18 is dedicated to this capability.

11.6 SUMMARY AND OUTCOMES OF CHAPTER 11

- The OS owns the supercomputer.
- An OS is a persistent program that controls the execution of application programs. It is the primary interface between user applications and system hardware.
- The primary function of the OS is to exploit the hardware resources of one or more processors, provide a set of services to system users, and manage secondary memory and I/O devices, including the file system.
- Resources managed by the OS are the processors and their integrated cores, the main memory of the systems out of which the cores work, I/O modules, and the system bus.
- User and system programs that are executing are made up of instances called "processes", which are instantiations of program procedures.
- Many processes may be operating concurrently under a single OS.
- OS memory management is responsible for address translation between the virtual addresses blocks of program data, called pages, and blocks of physical storage, called frames.
- The OS is responsible for the users' data and programs, organized in files through a hierarchy of named directories.
- The OS is responsible for managing all sources and destinations of data flow in and out of the computer it supports.
- The OS is responsible for a number of services associated with all the processes active in the system.
- At the heart of the process management services supported by the OS is the cross-cutting functionality of process scheduling: the determination of what processes are provided the necessary physical resources to run and when are they allocated.
- Threads present another level of parallelism control within a process.
- Virtual memory is a powerful abstraction for naming of memory blocks independent of their physical location, supported by the OS.
- The OS incorporates a page table, which is central to the method for associating virtual page addresses (or page numbers) to physical locations within the main memory or secondary storage.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 380

11.7 EXERCISES

1. Explain the differences between a TLB miss, a cache miss, and a page fault. What are the performance consequences of each?
2. What is the purpose of the virtual memory address?
3. What types of processes are kept in the ready queue? Which types of processes are not in the ready queue?
4. What is the difference between OS threads and processes? Can processes be preempted? Can threads be preempted? Are threads confined to a process?
5. Is the PCB affected by a thread context switch? Explain.

CHAPTER

VISUALIZATION

12

CHAPTER OUTLINE

12.1 Introduction 363
12.2 Foundational Visualization Concepts 364
12.3 Gnuplot 365
12.4 Matplotlib 369
12.5 The Visualization Toolkit 372
12.6 ParaView 379
12.7 VisIt 380
12.8 Summary and Outcomes of Chapter 12 381
12.9 Exercises 381
References 382

12.1 INTRODUCTION

Supercomputer applications frequently produce enormous amounts of output data that must be analyzed and presented to understand the application outcome and draw conclusions on the results. This process, frequently referred to as "visualization", can itself require supercomputing resources and is a fundamental modality of supercomputer usage.

Some of the principal reasons for visualizing data resulting from running an application on a supercomputer include debugging, exploring data, statistical hypothesis testing, and preparing presentation graphics. In some cases the output from running an application on a supercomputer will be something as simple as a single file with comma-separated values. However, it is much more likely that the output will be in a special parallel input/output (I/O) library format, like one of those mentioned in Chapter 10, to manage and coordinate the simultaneous output from multiple compute nodes to a single file.

This chapter discusses four key foundational concepts frequently needed as part of high performance computing (HPC) visualization: streamlines, isosurfaces, volume rendering through ray tracing, and mesh tessellations. Visualization is then practically explored through the use of five different visualization tools that are frequently used in the context of HPC: Gnuplot [1], Matplotlib [2], the Visualization Toolkit (VTK) library [3], ParaView [4], and VisIt [5]. Three of these tools (VTK, ParaView and VisIt) already incorporate the ability to use distributed memory parallel processing to accelerate the visualization process itself.

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00012-5
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 382

12.2 FOUNDATIONAL VISUALIZATION CONCEPTS

Among the most frequently used concepts in scientific visualization are streamlines, isosurfaces, volume rendering through ray tracing, and mesh tessellations. Streamlines, like those illustrated in Fig. 12.1, take a vector field as input and show curves that are tangent to the vector field. Streamlines may be thought of as showing the trajectory that a massless particle would travel in the input vector field. While the starting point for each streamline can be specified explicitly, it is more common to use random starting points seeded inside a small geometric object like a sphere or a cube.

An isosurface, illustrated in Fig. 12.2, is a surface that connects points which have the same value. Isosurfaces are frequently used in medical visualization to extract surfaces that have the same density, like that seen in a 3D ultrasound. An isosurface is the 3D analogue to a contour line in two-dimensional (2D) visualizations.

Volume rendering through ray tracing, illustrated in Fig. 12.3, is where for each pixel a ray is used to sample the volume through which it passes. Based on a provided color transfer function, the ray is shaded while an opacity function alters the transparency of the data in the volume. This type of volume rendering can reveal internal structures in data, and produce blurry or sharp edges depending on the opacity function chosen.

A mesh tessellation, seen in Fig. 12.4, is where a collection of data points and their connectivities to other data points are visualized through a set of polygons, frequently triangles or quadrilaterals in 2D and tetrahedra or hexahedra in 3D. The meshes often provide important statistical information about a simulation, including error bounds and mesh adaptivity, while also visually conveying the scale at which simulation features are resolved.

These foundational visualization concepts are usually not implemented directly by the application developer, but rather accessed in the context of an existing visualization toolkit or library. Some of the most common visualization toolkits and libraries for HPC are discussed in the following sections.

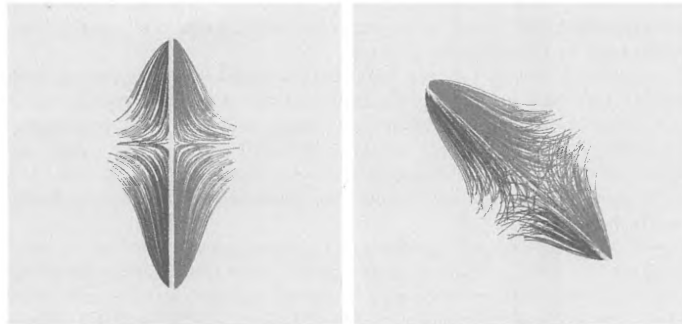


FIGURE 12.1

Streamline example using the gradient of the function $f(x,y,z) = 2550 \sin(100x) \sin(30y) \cos(40z)$ as input. Two different three-dimensional (3D) views are provided.

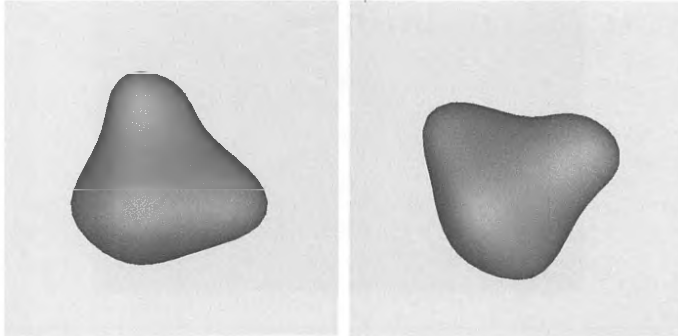


FIGURE 12.2

Isosurface example of the function $f(x,y,z) = 2550 \sin(10x) \sin(10y) \cos(10z)$ as input with the isosurface value set at 200. Two different 3-D views are provided.

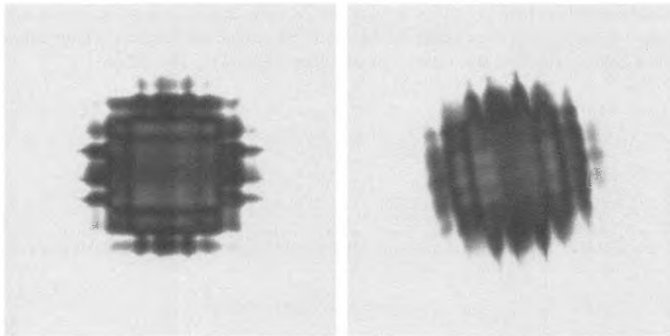


FIGURE 12.3

Example of low-resolution volume rendering of the function $f(x,y,z) = 2550 \sin(50x) \sin(50y) \cos(50z)$. The color and opacity map were chosen arbitrarily. Two different 3D views are provided.

12.3 GNUPLOT

Gnuplot [1] is a freely available and open-source command-line-driven visualization tool that includes support for both 2D and 3D plots. It has been around since 1986 and is found in most Linux distributions and on supercomputer login nodes. Several other independent applications use Gnuplot for

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 384

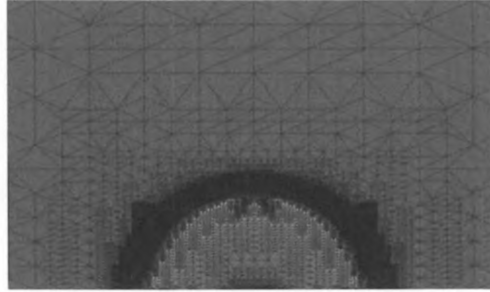


FIGURE 12.4

An example of a 2D mesh tessellation for an adaptive mesh shockwave simulation. The mesh, consisting of black lines, is visualized on top of a 2D color plot of the shockwave density.

graphics output, including GNU Octave [6], which features a high-level programming language very similar to Matlab [7].

Like most spreadsheet tools, Gnuplot is capable of a wide range of 2D plots. This is demonstrated here using the space-separated text data in Fig. 12.5. To initiate an interactive Gnuplot session, the *gnuplot* executable is launched from the command line, as shownin Fig. 12.6.

```
1 1 -1
2 2 -2
3 3 -3
4 4 -7
```

FIGURE 12.5

Example of three-column space-separated text data, referred to in the code examples as "*gnu_example.dat*".

```
Matthews-MacBook-Pro-2:data andersmw$ gnuplot

        G N U P L O T
        Version 5.0 patchlevel 5    last modified 2016-10-02

        Copyright (C) 1986-1993, 1998, 2004, 2007-2016
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type set to 'aqua'
gnuplot>
```

FIGURE 12.6

Launching an interactive Gnuplot session.

The *plot* command is the main command for 2D plots in Gnuplot, and takes the form of:

```
plot [ranges] <plot member> [, <plot member>, <plot member>]
```

If no ranges are specified, a default is computed based on the specific plot member. A plot member may be a predefined function like sin(x) or data read from a file, like that given in Fig. 12.5. Each plot member may have its plotting style altered using a predefined plotting style, such as *linespoints* or *circles*. Referring to the data in Fig. 12.5 as the file called "*gnu_example.dat*", three different ways of plotting with Gnuplot are illustrated in Figs. 12.7–12.9.

```
plot "gnu_example.dat" using 1:2 with linespoints
```

```
plot "gnu_example.dat" using 3:2 with linespoints
```

```
plot [0:4][-5:5] "gnu_example.dat" using 1:2 with linespoints title "data", sin(x)
title "sin(x)"
```

Gnuplot is also capable of 3D plots using the *splot* command, which shares most of the syntax of the 2D *plot* command. When plotting space-separated text data like that in Fig. 12.5, the first column gives the x values, the second the y values, and the third column is the value of the function at that point. An example of this is illustrated in Fig. 12.10.

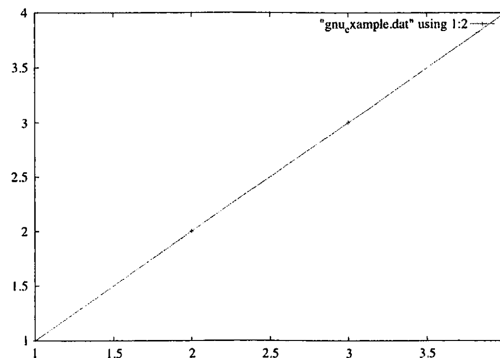


FIGURE 12.7

The first column of the data in Fig. 12.5 is used as the x values and the second column as the y values. Default ranges are generated.

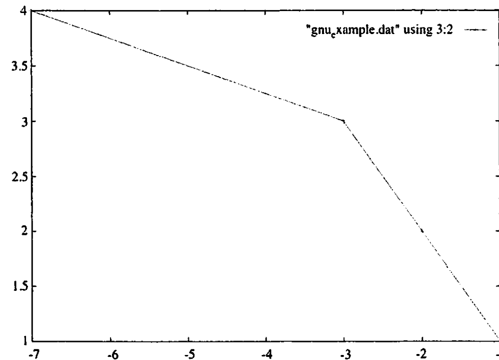


FIGURE 12.8

The third column of the data in Fig. 12.5 is used as the *x* values and the second column as the *y* values. Default ranges are generated.

```
splot "gnu_example.dat" with linespoints title "data", 10*exp(-(x-3)**2-(y-3)**2)
title "gaussian"
```

Among the many strengths of Gnuplot is the easy-to-use documentation accessed via the *help* command in interactive mode.

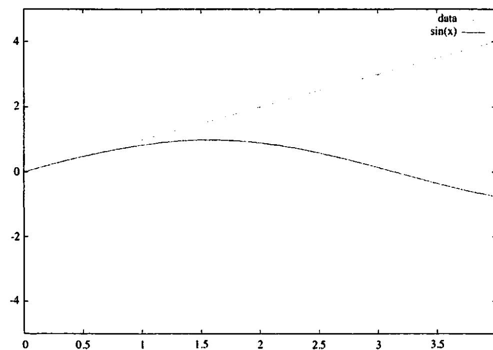


FIGURE 12.9

Plot containing the data from Fig. 12.5 as well as a plot of sin(*x*) with specified ranges.

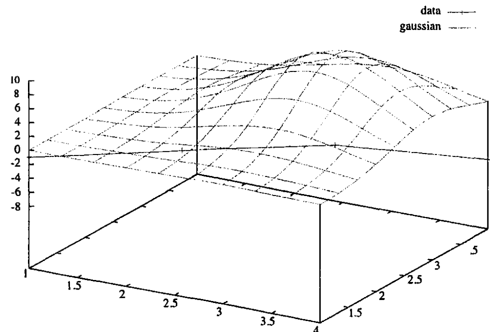


FIGURE 12.10

An example of a Gnuplot-generated 3D plot showing both the data in Fig. 12.1 and $f(x,y) = 10e^{-(x-3)^2-(y-3)^2}$.

12.4 MATPLOTLIB

Matplotlib [2] is a freely available and open-source Python language-based visualization tool with an interface that is similar to the look and feel of Matlab. It relies upon the NumPy extension to Python as a required dependency for array and matrix support. Like Gnuplot, Matplotlib is frequently found already installed on many supercomputers and is easily integrated into existing HPC application code bases for application-specific visualizations. Python is frequently used in scientific visualization, and in the case of Matplotlib using Python is a requirement. While the Python syntax is fairly simple and intuitive, a quick overview is given in an online guide [8].

In interactive mode, Matplotlib is initialized by launching Python and loading NumPy and Matplotlib, as illustrated below:

```
$ python
>>> import matplotlib.pyplot as plot
>>> import numpy
```

Once Matplotlib has been started in interactive mode, the data in Fig. 12.5 can be plotted interactively in a way analogous to that used with Gnuplot in Fig. 12.7. The Matplotlib example is shown in Fig. 12.11.

```
>>> data = numpy.loadtxt("gnu_example.dat",skiprows=0)
>>> xvalues = data.T[0]
>>> yvalues = data.T[1]
>>> l1, = plot.plot(xvalues,yvalues)
>>> plot.show()
```

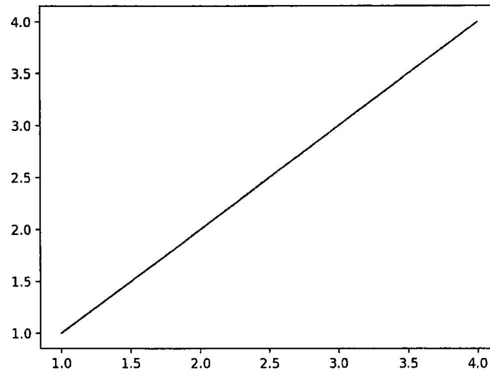


FIGURE 12.11

An interactive plot of the first and second columns of the data in Fig. 12.5 using Matplotlib. This is analogous to the Gnuplot version shown in Fig. 12.7. The text file "*gnu_example.dat*" is read into the data variable using the loadtxt method from NumPy. The data is rotated upon read-in so the data variable is transposed using the "T" operation and the columns are loaded into variables *xvalues* and *yvalues* for plotting.

Matplotlib easily integrates with the data-storage libraries explored in Chapter 10, including HDF5 and netCDF through their respective Python bindings. Data can then be easily manipulated using NumPy and plotted using Matplotlib. This is illustrated in Fig. 12.12.

In Fig. 12.12 the HDF5 dataset that was illustrated in Chapter 10 Fig. 10.7, *particles.h5*, is read into Python and the x and y values of the particles are plotted using Matplotlib. To do this, the Python bindings to HDF5 are loaded using the *import h5py* command in addition to loading Matplotlib and NumPy, as illustrated in the python script in Code 12.1.

```
1 import h5py
2 import numpy as np
3 import matplotlib.pyplot as plot
4
5 f = h5py.File("particles.h5","r")
6 dataset = f["particle data"]
7 xvalues = np.zeros(dataset.shape) #initializing memory
8 yvalues = np.zeros(dataset.shape) #initializing memory
9 for idx,item in enumerate(dataset):
10   xvalues[idx] = item[0]
11   yvalues[idx] = item[1]
12
13 l1, = plot.plot(xvalues,yvalues)
14 plot.show()
```

Code 12.1. Python code to plot the x and y values of the particle data stored in the "*particles.h5*" file created in Chapter 10.

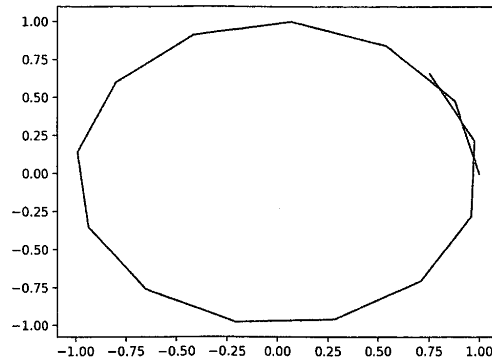


FIGURE 12.12

Matplotlib plotting the x and y coordinates of the particles data written in HDF5 format in Fig. 10.7 of Chapter 10. Matplotlib integrates well with the parallel I/O libraries discussed in Chapter 10.

The HDF5 file can then be loaded using the *h5py.File* method. A specific dataset in the file can be accessed by using the dataset name as a key to the file; in this case the dataset name is "particle data". A list of all datasets present in an HDF5 file can be found using the *h5ls* utility as well. The values in the dataset are copied to the appropriate *xvalues* and *yvalues* NumPy arrays and plotted, just as was done in Fig. 12.11.

Like Matlab, Matplotlib provides a number of tools for visualizing sparse matrices. One of the most common of these is the ability to plot the sparsity pattern of a matrix. This is illustrated in Code 12.2 and Fig. 12.13 for the matrix "*bcspwr06.mtx*" from the Matrix Market collection [9], using the matrix market reader provided in the SciPy ecosystem [10].

```
1 import scipy.io as sio
2 from matplotlib.pyplot import figure, show
3 import numpy
4
5 A = sio.mmread("bcspwr06.mtx");
6
7 fig = figure()
8 ax1 = fig.add_subplot(111)
9
10 ax1.spy(A,markersize=1)
11 show()
```

Code 12.2. Python script illustrating the ability to plot the sparsity pattern of a matrix. The matrix in this case, *bcspwr06.mtx*, comes from the Matrix Market collection [9]. The resulting sparsity pattern plot, producing using the spy method in line 8, is shown in Fig. 12.13.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 390

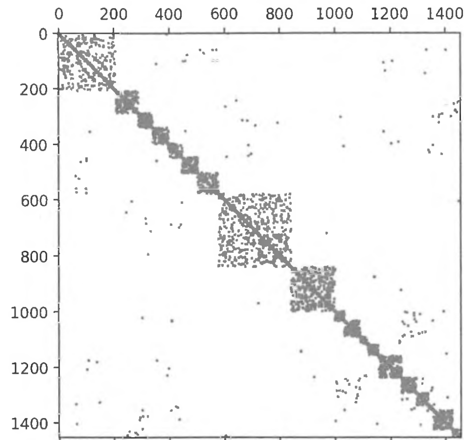


FIGURE 12.13

The sparsity pattern of a matrix plotted using Code 12.2.

Unlike Gnuplot, Matplotlib itself does not support 3D surface plots or other 3D-type visualizations. However, there are extension modules that can enable 3D plotting using Matplotlib, including mplot3d [11]. Matplotlib plots are also capable of integration within one of the most important and widely used libraries for 3D computer graphics, the VTK.

12.5 THE VISUALIZATION TOOLKIT

One of the most important open-source visualization libraries for HPC users is the VTK [3]. The VTK provides many 3D visualization algorithms, parallel computing support, and interfaces to interpreted languages like Python, which are used as examples in this section. The VTK is also used in several full visualization tools, including ParaView and VisIt, which are illustrated later in this chapter.

The most recent release of VTK is 8.0 and is conceptually laid out around the idea of a data pipeline incorporating maps with keys and values for passing information through the pipeline, objects for storing source data, algorithms, and filters, and a class for connecting together and executing the pipeline. In VTK terminology, "mappers" convert data into graphics primitives while "actors" alter the visual properties of those graphics. The example shown in Fig. 12.14 and Code 12.3 reads the HDF5 data "*particles.h5*" from Chapter 10 and plots a line in 3D through the points in the HDF5 dataset using VTK.

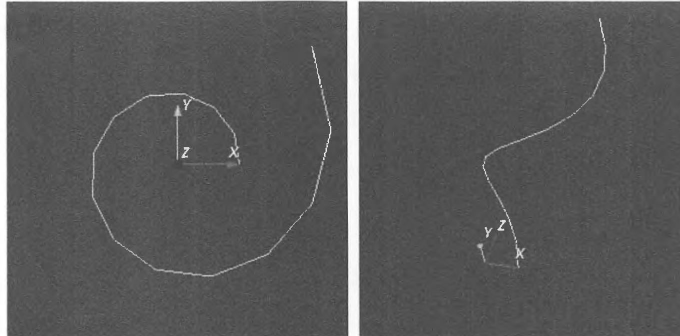


FIGURE 12.14

Two 3D plots of the particle locations found in *particles.h5* from Chapter 10 using VTK. The corresponding code for this visualization is found in Code 12.3.

```
1 import h5py     # the HDF5 Python interface
2 import vtk      # the VTK Python interface
3 f = h5py.File("particles.h5","r")  # read in the file "particles.h5"
4 dataset = f["particle data"]  # access the dataset "particle data" in "particles.h5"
5 points = vtk.vtkPoints()
6 points.SetNumberOfPoints(dataset.shape[0])  # create a list of points for
  particle locations
7 for idx,item in enumerate(dataset):
8 points.SetPoint(idx,dataset[idx][0],dataset[idx][1],dataset[idx][2]) # assign
  values
9
10 lines = vtk.vtkCellArray()
11 lines.InsertNextCell(dataset.shape[0])
12 for idx in range(0,dataset.shape[0]):  # assign the connectivity between the
   points
13 lines.InsertCellPoint(idx)
14
15 polygon = vtk.vtkPolyData()  # create a polygon geometric structure
16 polygon.SetPoints(points)
17 polygon.SetLines(lines)
18
19 polygonMapper = vtk.vtkPolyDataMapper()  # map the polygonal data to graphics
20 polygonMapper.SetInputData(polygon)
21 polygonMapper.Update()
22
23 axes = vtk.vtkAxesActor()    # create some axes
24 polygonActor = vtk.vtkActor()    # Manage the rendering of the mapper
25 polygonActor.SetMapper(polygonMapper)
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 392

```
26 renderer = vtk.vtkRenderer()     # The viewport on the screen
27 renderer.AddActor(polygonActor)
28 renderer.AddActor(axes)
29 renderer.SetBackground(0.1, 0.2, 0.3)
30
31 renderer.ResetCamera()
32
33 renderWindow = vtk.vtkRenderWindow()
34 renderWindow.AddRenderer(renderer)
35
36 interactive_ren = vtk.vtkRenderWindowInteractor()  # enable interactivity with
   visualization
37 interactive_ren.SetRenderWindow(renderWindow)
38 interactive_ren.Initialize()
39 interactive_ren.Start()
```

Code 12.3. A Python script to read in and visualize the 3D trajectory of the particle data stored in *particles.h5* from Chapter 10 using VTK. The resulting visualization is seen in Fig. 12.14.

All other major scientific visualization components are available in VTK. Isosurfaces of 3D data can be produced using vtkContourFilter, as illustrated in Fig. 12.15. In VTK, filters like vtkContourFilter are optionally applied in the pipeline before applying mappers and actors, as illustrated in Code 12.4.

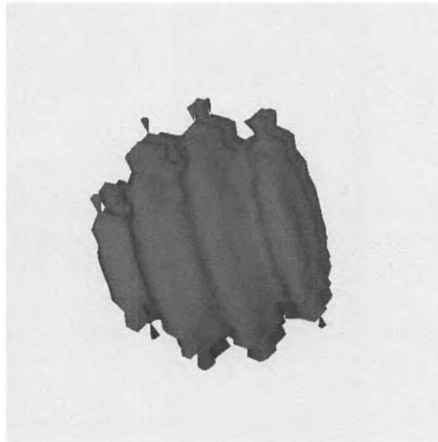


FIGURE 12.15

An isosurface in VTK using Code 12.4.

```
1 import vtk # the VTK Python interface
2
3 rt = vtk.vtkRTAnalyticSource()  # data for testing
4
5 contour_filter = vtk.vtkContourFilter()  # isosurface filter
6 contour_filter.SetInputConnection(rt.GetOutputPort())
7 contour_filter.SetValue(0, 190)
8
9 mapper = vtk.vtkPolyDataMapper()
10 mapper.SetInputConnection(contour_filter.GetOutputPort())
11
12 actor = vtk.vtkActor()
13 actor.SetMapper(mapper)
14
15 renderer = vtk.vtkRenderer()
16 renderer.AddActor(actor)
17
18 renderer.SetBackground(0.9, 0.9, 0.9)
19
20 renderWindow = vtk.vtkRenderWindow()
21 renderWindow.AddRenderer(renderer)
22 renderWindow.SetSize(600, 600)
23
24 interactive_ren = vtk.vtkRenderWindowInteractor()  # enable interactivity with
   visualization
25 interactive_ren.SetRenderWindow(renderWindow)
26 interactive_ren.Initialize()
27 interactive_ren.Start()
```

Code 12.4. Example isosurface using the ContourFilter filter; the value of the isosurface is set at line 7. Test data was provided using vtkRTAnalyticSource in line 3. The resulting visualization is shown in Fig. 12.15.

One way to execute volume rendering through ray tracing in VTK is using the SmartVolumeMapper class illustrated in Code 12.5 and Fig. 12.16. In this example, a color transfer function and an opacity map are passed as properties to shade the rays appropriately as they pass through the volume.

```
1 import vtk
2
3 rt = vtk.vtkRTAnalyticSource()
4 rt.Update()
5
6 image = rt.GetOutput()
7 range = image.GetScalarRange()
8
9 mapper = vtk.vtkSmartVolumeMapper()   # volume rendering
10 mapper.SetInputConnection(rt.GetOutputPort())
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 394

```
11 mapper.SetRequestedRenderModeToRayCast()
12
13 color = vtk.vtkColorTransferFunction()
14 color.AddRGBPoint(range[0], 0.0, 0.0, 1.0)
15 color.AddRGBPoint((range[0] + range[1]) * 0.75, 0.0, 1.0, 0.0)
16 color.AddRGBPoint(range[1], 1.0, 0.0, 0.0)
17
18 opacity = vtk.vtkPiecewiseFunction()
19 opacity.AddPoint(range[0], 0.0)
20 opacity.AddPoint((range[0] + range[1]) * 0.5, 0.0)
21 opacity.AddPoint(range[1], 1.0)
22
23 properties = vtk.vtkVolumeProperty()
24 properties.SetColor(color)
25 properties.SetScalarOpacity(opacity)
26 properties.SetInterpolationTypeToLinear()
27 properties.ShadeOn()
28
29 actor = vtk.vtkVolume()
30 actor.SetMapper(mapper)
31 actor.SetProperty(properties)
32
33 renderer = vtk.vtkRenderer()
34 renderWindow = vtk.vtkRenderWindow()
35 renderWindow.AddRenderer(renderer)
36
37 renderer.AddViewProp(actor)
38 renderer.ResetCamera()
39 renderer.SetBackground(0.9, 0.9, 0.9)
40 renderWindow.SetSize(600, 600)
41
42 interactive_ren = vtk.vtkRenderWindowInteractor()
43 interactive_ren.SetRenderWindow(renderWindow)
44 interactive_ren.Initialize()
45 interactive_ren.Start()
```

Code 12.5. Example volume rendering using VTK. Test data was provided using vtkRTAnalyticSource at line 3. Opacity and color map settings were made based on the image scalar range. The resulting visualization is shown in Fig. 12.16.

Streamlines in VTK are accomplished using the StreamTracer class. Streamlines require vector data as input, but VTK also provides a means to take a gradient of scalar data and then assign output as a vector which can be visualized as a streamline. This entire pipeline is demonstrated in Code 12.6 and Fig. 12.17. The starting point for a single streamline can be specified, as illustrated in the comment on line 31 of Code 12.6, or a streamline seed region can be created for starting multiple streamlines, as illustrated in lines 23–27.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 395

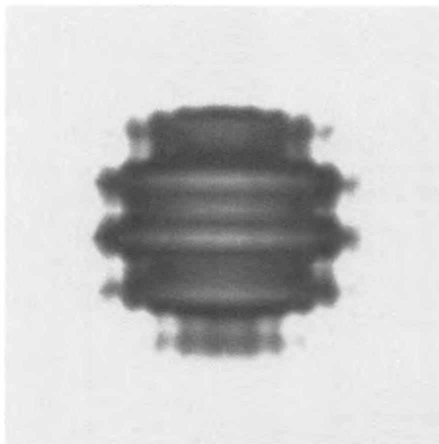


FIGURE 12.16

An example volume rendering in VTK using Code 12.5.

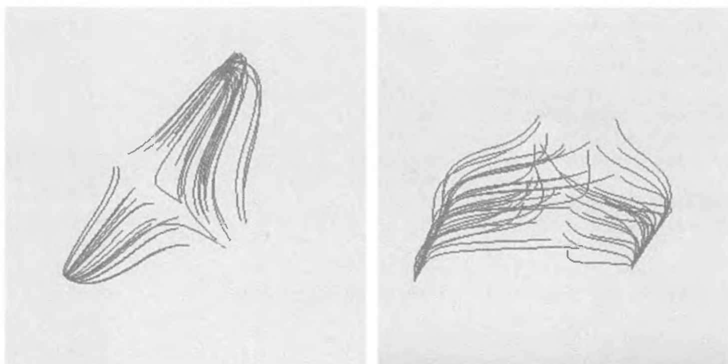


FIGURE 12.17

Streamlines in VTK using the gradient of the data shown in Figs. 12.15 and 12.16. The code that produced these streamlines in VTK is shown in Code 12.6.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 396

```
1 import vtk
2
3 rt = vtk.vtkRTAnalyticSource()  # data for testing
4 rt.Update()
5
6 #calculate the gradient of the test data
7 gradient = vtk.vtkImageGradient()
8 gradient.SetDimensionality(3)
9 gradient.SetInputConnection(rt.GetOutputPort())
10 gradient.Update()
11
12 # Make a vector
13 aa = vtk.vtkAssignAttribute()
14 aa.Assign("SCALARS","VECTORS","POINT_DATA")
15 aa.SetInputConnection(gradient.GetOutputPort())
16 aa.Update()
17
18 # Create Stream Lines
19 rk = vtk.vtkRungeKutta45()
20 streamer = vtk.vtkStreamTracer()
21 streamer.SetInputConnection(aa.GetOutputPort())
22
23 # seed the stream lines
24 seeds = vtk.vtkPointSource()
25 seeds.SetRadius(1)
26 seeds.SetCenter(1,1.1,0.5)
27 seeds.SetNumberOfPoints(50)
28
29 # options for streamer
30 streamer.SetSourceConnection(seeds.GetOutputPort())
31 #streamer.SetStartPosition(1.0,1.1,0.5)
32 streamer.SetMaximumPropagation(500)
33 streamer.SetMinimumIntegrationStep(0.01)
34 streamer.SetMaximumIntegrationStep(0.5)
35 streamer.SetIntegrator(rk)
36 streamer.SetMaximumError(1.0e-8)
37
38 mapStream = vtk.vtkPolyDataMapper()
39 mapStream.SetInputConnection(streamer.GetOutputPort())
40 streamActor = vtk.vtkActor()
41 streamActor.SetMapper(mapStream)
42
43 ren = vtk.vtkRenderer()
44 renWin = vtk.vtkRenderWindow()
45 renWin.AddRenderer(ren)
46 iren = vtk.vtkRenderWindowInteractor()
47 iren.SetRenderWindow(renWin)
```

```
48
49 ren.AddActor(streamActor)
50 ren.SetBackground(0.9,0.9,0.9)
51 renWin.SetSize(300,300)
52 iren.Initialize()
53 iren.Start()
```

Code 12.6. Example code using vtkRTAnalyticSource to create streamlines with VTK. The gradient of the test data is computed in lines 7–10; the gradient is then assigned as vector data for use by the vtkStreamTracer filter for producing streamlines. The starting points for the streamlines are produced from point sources in a sphere computed in lines 24–27. A single starting point could also be assigned, as illustrated in the comment in line 31. The result from this code is seen in Fig. 12.17.

While the VTK library provides a complete visualization pipeline solution for HPC users, many users will prefer a turnkey visualization solution that is driven by a powerful graphical user interface (GUI) and is ready for supercomputing usage without having to write any code. Two widely used turnkey visualization tools that incorporate the powerful algorithms of VTK are ParaView and VisIt.

12.6 PARAVIEW

ParaView is an open-source HPC-capable turnkey visualization solution based on VTK. Like other visualization tools examined in this chapter, significant support for the Python language is provided, enabling control of ParaView from both a GUI or a script. Because ParaView is based on VTK, the naming of elements in the visualization pipeline follows that of the VTK API. ParaView has data readers for over 70 different data formats. An example dataset that comes with ParaView is shown in Fig. 12.18.

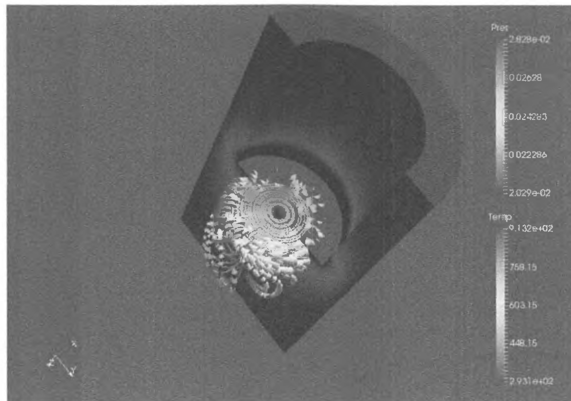


FIGURE 12.18

Example visualization that comes with ParaView illustrating streamlines with arrows and data slices.

12.7 VISIT

VisIt is another open-source HPC-capable turnkey visualization solution that uses VTK for several visualization algorithms. VisIt is particularly well suited for *in situ* visualization which occurs while the supercomputing simulation that creates the data is ongoing. An example VisIt visualization is shown in Fig. 12.19, with a skewed color map to reveal features in the data that would not be otherwise apparent.

VTK accepts over a 100 different data input formats and provides a simple scripting interface as an alternative to using the GUI for creating visualization.

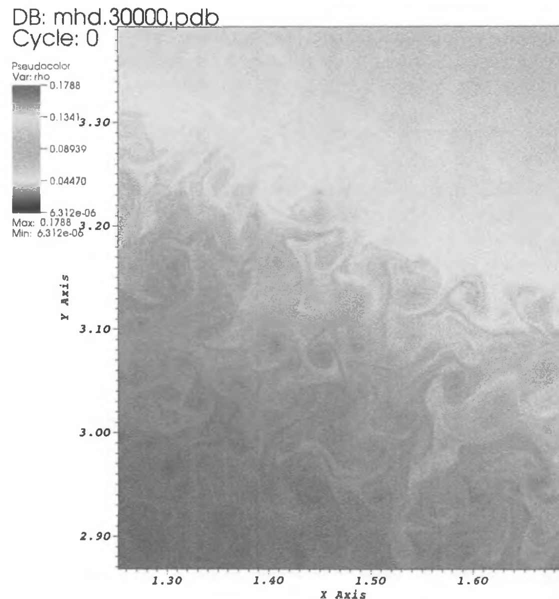


FIGURE 12.19

Example VisIt pseudo-color plot using a skewed color map to reveal a physical instability manifested as rolls in the data. The name of the visualized file appears in the upper left-hand corner, and the field variable name ("rho") is given above the color legend.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 399

12.8 SUMMARY AND OUTCOMES OF CHAPTER 12

- Motivations for visualizing data include debugging, exploring data, statistical hypothesis testing, and preparing presentation graphics.
- Many scientific visualizations incorporate at least one of four foundational visualization concepts: streamlines, isosurfaces, volume rendering by ray tracing, and mesh tessellations.
- Streamlines take a vector field as input and show curves that are tangent to the vector field
- Isosurfaces are surfaces connecting data points which have the same value.
- Volume rendering by ray tracing casts rays through the data volume and samples the volume through which the rays pass.
- Mesh tessellations visualize data points and their connectivities to other data points using polygons.
- Gnuplot is a simple command-line visualization tool for 2D and 3D plots.
- Matplotlib is a Python-based visualization tool with easy integration to other libraries with Python bindings.
- VTK is an open-source collection of visualization algorithms for creating application-specific visualization solutions.
- ParaView and VisIt are turnkey visualization solutions incorporating VTK algorithms but providing a GUI and scripting interface for visualization.
- ParaView and VisIt already incorporate support for hundreds of widely used data formats on HPC systems.

12.9 EXERCISES

1. List all the factors that impact a decision to use a particular visualization approach for an HPC application. Create a table listing the trade-off space of the five visualization tools explored in this chapter.
2. Create a set of streamlines, isosurfaces, and volume renderings of the function $f(x,y,z) = 2550 \sin(50x) \sin(50y) \cos(50z)$ using the visualization tool of your choice.
3. Create a 2D dataset using the output library of your choice (i.e., HDF5, NetCDF, Silo, etc.) for the following function: $f(x,y) = e^{-x^2-y^2}$ where $x \in [-1, 1]$ and $y \in [-1, 1]$. Then, using the visualization library of your choice, read in this data and visualize it. Finally, compute the gradient of this data using the visualization tool you have chosen, and plot the result.
4. Visualization tools provide a large number of optional color legends. Why? In what circumstances is one color legend better than another?
5. Explore the parallel visualization capabilities of VisIt or Paraview by redoing problem 2 but using HPC resources. Produce a strong scaling plot showing the time to solution for the visualization as a function of the number of computing resources employed.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 400

REFERENCES

1. Gnuplot homepage [Online], http://www.gnuplot.info/.
2. Matplotlib [Online], http://matplotlib.org/.
3. The Visualization Toolkit [Online], http://www.vtk.org/.
4. ParaView [Online], http://www.paraview.org/.
5. VisIt [Online], https://wci.llnl.gov/simulation/computer-codes/visit.
6. GNU Octave [Online], https://www.gnu.org/software/octave/.
7. MathWorks, MATLAB [Online], https://www.mathworks.com/products/matlab.html.
8. Python.org, Python Beginners Guide [Online], https://wiki.python.org/moin/BeginnersGuide/Programmers.
9. National Institute of Standards and Technology, Martrix Market Collection [Online], http://math.nist.gov/MatrixMarket/.
10. SciPy Developers, SciPy [Online], https://www.scipy.org/.
11. Matplotlib, mplot3d [Online], http://matplotlib.org/mpl_toolkits/mplot3d/.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 401

CHAPTER

PERFORMANCE MONITORING 13

CHAPTER OUTLINE

13.1 Introduction 383
13.2 Time Measurement 385
13.3 Performance Profiling 390
13.3.1 Significance of Application Profiling 390
13.3.2 Essential *gperftools* 391
13.4 Monitoring Hardware Events 398
13.4.1 Perf 398
13.4.2 Performance Application Programming Interface 404
13.5 Integrated Performance Monitoring Toolkits 407
13.6 Profiling in Distributed Environments 411
13.7 Summary and Outcomes of Chapter 13 417
13.8 Questions and Problems 418
References 419

13.1 INTRODUCTION

Performance monitoring is both an inherent and a key step in application development. The code development process does not stop when the program appears to be doing what it was designed to do and the generated results are validated for correctness. Even when the application has been tested using a broad range of input parameters and datasets as well as multiple supported computational modes stressing individual program features, there may still be hidden problems preventing it from executing at the smaximum performance permitted by the underlying platform. This is particularly important in parallel computing, where the impact of every inefficiency is effectively multiplied by the number of processor cores the application is running on. Besides increasing the time necessary to arrive at solutions, this may also have financial implications, since often the user is charged for computer use in proportion to the consumed aggregate machine time. One of the most important reasons for performance monitoring is therefore to verify that the application is not impacted by any obvious or easily preventable degradation factors. One way to confirm this is a simple sanity check: is the actual computation time in line with the processor speed and the estimated total number of operations that need to be performed? Is the communication phase taking longer than estimated given the message sizes transmitted by the application and network bandwidth? Fortunately, these questions may be

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00013-7
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 402

answered by performing simple instrumentation of application code to measure the time required to execute the segments of the program involved. This is discussed in Section 13.2.

Even a simple measurement, such as capture of a timestamp, affects program execution due to greater than zero latency of the operation accessing the system timer and the nonzero resource footprint required to perform the operation. The more complex and frequent the measurements, the more overhead is introduced into program execution, potentially skewing the measurement results and in the worst case completely changing the application execution flow. The latter may be particularly damaging, since the identification of sections of code that need to be revised for performance improvement may be incorrect and cause additional programmer effort with little or no benefit. One way to alleviate this problem is to apply statistical sampling. Instead of registering every occurrence of an event in a program, a snapshot of program's state (sample) is taken at fixed intervals. The sampling period usually may be raised within a permitted range to increase the accuracy (again, at the cost of additional overhead) when there is a good possibility that some events were not accounted for, or lowered if the monitoring is discovered to be too intrusive or a coarse execution profile is sufficient. Another, albeit limited, way to lower instrumentation overheads is to use custom hardware to capture the events of interest. Modern CPUs implement dedicated registers that may be configured to count the occurrences of specific low-level events, such as branches, cache misses, instruction retirement, etc. Since the register updates are carried out entirely in hardware, executing software almost never sees the monitoring overhead. However, the consequence of hardware implementation is that the classes of supported events are predefined and cannot be extended or customized.

The remainder of this chapter discusses various performance monitoring tools commonly used to evaluate high performance computing workloads. Due to easier accessibility, broader portability, and no licensing costs, open-source tools are usually preferred. However, there are several useful proprietary tools that provide easy-to-use interfaces (especially those driven by a graphic user interface or GUI) and may leverage better technical expertise of hardware products than can be derived from publicly available documentation. While they will not be discussed in depth, they deserve a mention and are listed here to make the reader aware of other performance analysis options:

- Intel VTune Amplifier [1] is an integrated profiling environment targeting primarily Intel CPUs, including Xeon Phi. It can perform statistical hotspot analysis, thread profiling, and lock and blocking analysis, measure floating-point unit (FPU) utilization and Flops values, analyze memory and storage accesses, and trace computation offload to Graphics Processing Units (GPUs) via OpenCL. The tool integrates with Intel Parallel Studio XE and Microsoft Visual Studio, and supports programming languages such as C, C++, C#, Fortran, Java, Python, Go, and assembly. It is also capable of remote trace collection to enable monitoring of distributed applications such as a message-passing interface (MPI). Supported operating systems include Linux, Windows, and Mac OS X.
- CodeXL [2] is AMD's equivalent of VTune, providing an integrated suite of tools for performance analysis targeting x86-compatible CPUs as well as AMD GPUs and accelerated processing units (APUs) through the OpenCL Software Development Toolkit (SDK). It supports time-based profiling on CPUs, event-based profiling and instruction-based sampling on CPUs and APUs, and real-time power profiling including capture of CPU core clock frequencies, thermal trends, and P-states. CodeXL may be used as a standalone tool on Linux (Red Hat, Ubuntu, SUSE) and Windows, and is also available as an extension to Microsoft Visual Studio. While the source code is available through GitHub [3], much of the tool's core functionality relies on the proprietary AMD Catalyst software [4].

- The Nvidia CUDA Toolkit [5] includes a visual profiler (*nvvp*) that can be used to monitor and analyze the execution of parallel programs on Nvidia GPUs. Through collected traces, it gives the user an insight into program activity and the execution timeline decomposed into individual processing threads and workload phases. It also monitors memory usage (including unified memory on supporting architectures) as well as power consumption, clock speed, and thermal conditions. The tool has an option to analyze Pthread behavior on the host CPU as well as OpenACC applications (this requires a PGI compiler). Profiling may also be enabled from the command line using the *nvprof* utility. The toolkit is available for Linux, OS X, and Windows.

13.2 TIME MEASUREMENT

Execution time is one of the critical metrics of application performance and of primary importance to both application developers and end users. Its measurement may be carried out at the whole-program level as well as for selected sections of the monitored application. Each of these scenarios requires a different approach. The measurement of duration of application execution should typically be synchronized with the wall clock time to establish a common reference permitting meaningful comparisons with results obtained on other platforms and environments. This is particularly important when application execution takes a significant amount of time, potentially counted in days or months. Most computer system clocks are periodically synchronized over the network to a common high-accuracy standard, typically derived from an atomic clock using protocols such as Network Time Protocol (NTP) [6]. This provides sufficiently good average accuracy in the long term, although it does not avoid the issue of local clock jitter. It is also affected by the characteristics of the clock adjustment algorithm: if the measurement happens when the system clock's value is updated to match the standard, potentially a large skew may be introduced to the result. Most implementations tend to tune the system clock gradually by small amounts, thus alleviating the problem of a measured value being dependent on the time when the act of measurement is being performed.

Most Unix systems provide several utilities to access the wall clock time from the command line. One is the *date* program that outputs the current date and time with accuracy down to single seconds. It may be used in batch job scripts to provide coarse timestamps for the start and end times of application execution (or any intermediate phases as long as they are represented by separate applications). Its output will be captured in a file storing the standard output stream of the job's execution shell for future inspection. An example output of the command as invoked from the shell prompt is:

```
> date
Sun, Feb 05, 2017 6:17:33 PM
```

The date command also accepts custom date format specification as a command-line argument in case the default form shown above is not acceptable.

Since resolution at the full-second level may not be sufficient for short-running applications, more precise measurements can use the *time* utility that may be available as a bash shell built-in command or a standalone system program. It has to be followed by a correctly formed command line fully describing the application with its options and command-line arguments. The specified application

will immediately be spawned as specified, while the timing utility captures several key characteristics of its execution. For example:

```
> /usr/bin/time dd if=/dev/zero of=/dev/null bs=4096 count=1M
1048576+0 records in
1048576+0 records out
4294967296 bytes (4.3 GB, 4.0 GiB) copied, 0.482873 s, 8.9 GB/s
0.37user 0.10system 0:00.48elapsed 100%CPU (0avgtext+0avgdata 415744maxresident)k
0inputs+0outputs (1643major+0minor)pagefaults 0swaps
```

The above times the execution of the *dd* program (available on any Linux distribution and used to copy and convert file data) that transfers 4 GB of zero-filled data to a null device. Note that the first three lines contain output from the *dd* utility itself. In this case, the program execution took 0.48 s (as given by the elapsed time entry), of which 0.37 s were spent executing user code and 0.1 s system (or kernel) code. The reported system and user times do not necessarily have to add to the elapsed time value. This is because program execution may be stalled, e.g., waiting for user input, completion of input/output (I/O) operations, or other external events. If the program could not fully utilize the allocated processor core(s), the reported utilization (as a percentage of the CPU) may be lower than 100%. Note that multithreaded programs may report values greater than 100%, since the displayed user and system times are the aggregate values over all compute threads spawned by the application.

The time utility also reports other details of program execution that may be helpful in analyzing the application's behavior. One of them, following the timings, provides information about memory resources allocated by the application. The first number indicates the average size of memory used by program text (instruction pages), the second represents the average size of unshared program data, and the third shows the maximum size of physical memory (resident set) used by the application's process. These numbers are reported in kilobytes. The last line displayed by the time command lists the number of I/O operations performed by the program, the number of minor and major page faults, and how many times the process was swapped out from memory for disk. The difference between major and minor page faults is that the first involves access to a storage device required to retrieve the contents of memory page, while a minor fault only requires an update of a page table entry. Thus the cost of a major fault is typically substantially higher than that of a minor fault. Similarly to *date*, the output of the *time* command may be customized through the command-line option `-f` or `--format` to include additional parameters such as the number of involuntary and voluntary context switches, the number of messages in socket-based communication, the number of signals delivered to the process, and the exit status of the process. Note that the shell built-in *time* command reports only the user, system, and elapsed timings for the monitored program.

The timing utilities operating at the whole-application level are not useful for measuring duration of execution of individual functions or code segments. For that purpose, a number of timing functions accessing the system's high-resolution clock are used. Individual implementations of high-resolution timers may differ from system to system depending on the actual processor type and system configuration. Since the native interfaces exposed by such timers are often not compatible with each other, typically the most portable way to access them is to use POSIX clock functions. The most frequently

used call, clock_gettime, obtains the value of time that has elapsed from some fixed reference point in the past, typically machine boot time. Its prototype looks as follows:

```
#include <time.h>
int clock_gettime(clockid_t id, struct timespec *tsp);
```

where id identifies one of the clocks available on the system and the structure to store the time data pointed to by tsp comprises two fields. The first of them, tv_sec contains the number of full seconds and the other, tv_nsec, stores the number of nanoseconds expressing the remaining fraction of a second for the measured time interval. Both of these fields are integers of sufficient size to store the required data, frequently equal to machine's native word size. The function returns zero on success. To verify the actual resolution of the accessed clock, POSIX provides the clock_getres function that takes the clock identifier argument and stores the measurement resolution value in a structure pointed to by tsp:

```
int clock_getres(clockid_t id, struct timespec *tsp);
```

For fine-granularity measurements, useful clock ids include CLOCK_MONOTONIC and CLOCK_MONOTONIC_RAW. Unlike the system wall clock, which may be subjected to coarse changes of value due to the administrator manually adjusting the system time, the monotonic clock is only affected by incremental adjustments performed by the time synchronization protocol in use (e.g., NTP). The raw monotonic clock has the same properties as the monotonic clock, but it is not affected by external time adjustment. The POSIX interface also supports other clocks of interest: CLOCK_BOOTTIME that is similar to the monotonic clock, but includes the time that elapsed while the system was suspended; CLOCK_PROCESS_CPUTIME_ID, which measures processor time consumed by all threads in the process it was called in; and CLOCK_THREAD_CPUTIME_ID for a processor time clock that is limited to the specific thread. Selection of the suitable clock should be performed in the context of application and type of measurement; for most performance measurements on an "always-on" platform, the monotonic clock is often sufficient as long as the overhead of several tens of nanoseconds per access is acceptable.

To take advantage of POSIX clocks, the user code needs to be explicitly instrumented with timing functions. To demonstrate this, a program performing matrix–vector multiplication with source code listed in Code 13.1 is used. The same code is subsequently subjected to analysis by other performance monitoring tools in the next sections of this chapter. The application allocates heap memory, initializes the matrix and multiplicand and product vectors (routine init, lines 6–15), performs the multiplication by invoking the CBLAS library function (refer to Chapter 11 for more details on BLAS) to compute dot products (mult function, lines 17–23), and verifies the result by performing an absolute value sum on the elements of the product vector (cblas_dasum in line 33). Both initialization and multiplication can be performed in row- or column-major fashion, potentially impacting the duration of computations. This is controlled by the second command-line argument (transposition flag); the first one specifies the size of the matrix. To gather the timing information,

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 406

`clock_gettime` functions were added to the `main` function of the source as shown in Code 13.2 (only the instrumented section is provided; the starting part of the program preceding line 25 is unchanged). Code 13.2 also defines the `sec` function that is used to convert the contents of the `timespec` structure to a floating-point number of seconds, thus enabling a straightforward calculation of time intervals. Note that collection of timestamps is arranged with as little additional code as possible, and therefore the conversions of timing values to seconds and printout of final values are performed outside the timed regions.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <cblas.h>
 4 #include <time.h>
 5
 6 void init(int n, double **m, double **v, double **p, int trans) {
 7   *m = calloc(n*n, sizeof(double));
 8   *v = calloc(n, sizeof(double));
 9   *p = calloc(n, sizeof(double));
10   for (int i = 0; i < n; i++) {
11     (*v)[i] = (i & 1)? -1.0: 1.0;
12     if (trans) for (int j = 0; j <= i; j++) (*m)[j*n+i] = 1.0;
13     else for (int j = 0; j <= i; j++) (*m)[i*n+j] = 1.0;
14   }
15 }
16
17 void mult(int size, double *m, double *v, double *p, int trans) {
18   int stride = trans? size: 1;
19   for (int i = 0; i < size; i++) {
20     int mi = trans? i: i*size;
21     p[i] = cblas_ddot(size, m+mi, stride, v, 1);
22   }
23 }
24
25 int main(int argc, char **argv) {
26   int n = 1000, trans = 0;
27   if (argc > 1) n = strtol(argv[1], NULL, 10);
28   if (argc > 2) trans = (argv[2][0] == 't');
29
30   double *m, *v, *p;
31   init(n, &m, &v, &p, trans);
32   mult(n, m, v, p, trans);
33   double s = cblas_dasum(n, p, 1);
34   printf("Size %d; abs. sum: %f (expected: %d)\n", n, s, (n+1)/2);
35   return 0;
36 }
```

Code 13.1. Matrix—vector multiply code operating in row- and column-major modes.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 407

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <cblas.h>
 4 #include <time.h>
 5
 6 void init(int n, double **m, double **v, double **p, int trans) {
 7   *m = calloc(n*n, sizeof(double));
 8   *v = calloc(n, sizeof(double));
 9   *p = calloc(n, sizeof(double));
10   for (int i = 0; i < n; i++) {
11     (*v)[i] = (i & 1)? -1.0: 1.0;
12     if (trans) for (int j = 0; j <= i; j++) (*m)[j*n+i] = 1.0;
13     else for (int j = 0; j <= i; j++) (*m)[i*n+j] = 1.0;
14   }
15 }
16
17 void mult(int size, double *m, double *v, double *p, int trans) {
18   int stride = trans? size: 1;
19   for (int i = 0; i < size; i++) {
20     int mi = trans? i: i*size;
21     p[i] = cblas_ddot(size, m+mi, stride, v, 1);
22   }
23 }
24
25 double sec(struct timespec *ts) {
26   return ts->tv_sec+1e-9*ts->tv_nsec;
27 }
28
29 int main(int argc, char **argv) {
30   struct timespec t0, t1, t2, t3, t4;
31   clock_gettime(CLOCK_MONOTONIC, &t0);
32   int n = 1000, trans = 0;
33   if (argc > 1) n = strtol(argv[1], NULL, 10);
34   if (argc > 2) trans = (argv[2][0] == 't');
35
36   double *m, *v, *p;
37   clock_gettime(CLOCK_MONOTONIC, &t1);
38   init(n, &m, &v, &p, trans);
39   clock_gettime(CLOCK_MONOTONIC, &t2);
40   mult(n, m, v, p, trans);
41   clock_gettime(CLOCK_MONOTONIC, &t3);
42   double s = cblas_dasum(n, p, 1);
43   clock_gettime(CLOCK_MONOTONIC, &t4);
44   printf("Size %d; abs. sum: %f (expected: %d)\n", n, s, (n+1)/2);
45   printf("Timings:\n program: %f s\n", sec(&t4)-sec(&t0));
46   printf("  init: %f s\n  mult: %f s\n  sum: %f s\n",
47          sec(&t2)-sec(&t1), sec(&t3)-sec(&t2), sec(&t4)-sec(&t3));
48   return 0;
49 }
```

Code 13.2. Instrumented section of the matrix multiplication code.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 408

Execution of the instrumented code in row-major mode with matrix size 10,000 × 10,000 yields:

```
> ./mvmult 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)
Timings:
  program: 1.148853 s
     init: 0.572537 s
     mult: 0.576276 s
      sum: 0.000037 s
```

Doing the same using the less efficient column-major operation results in:

```
> ./mvmult 20000 t
Size 20000; abs. sum: 10000.000000 (expected: 10000)
Timings:
  program: 12.126625 s
     init: 4.343727 s
     mult: 7.782852 s
      sum: 0.000043 s
```

As can be seen, program execution in the transposed mode takes an order of magnitude longer. The change is attributed primarily to a substantial increase in execution time of initialization and multiplication subroutines that access the matrix data. The absolute sum performed in the verification phase takes roughly the same amount of time, since the layout of the input data (product vector) does not change.

13.3 PERFORMANCE PROFILING

13.3.1 SIGNIFICANCE OF APPLICATION PROFILING

The goal of profiling is to provide an insight into application execution that may help identify the potential performance problems. These may be related to the algorithmic code makeup, memory management, communication, or I/O. Profiling tools usually concentrate on *hotspot* analysis—that is, detection of the parts of code the program spends most of its time executing. This may lead to identification of bottlenecks, or those hotspots that have unduly adverse effects on the application's performance. A bottleneck is usually apparent as a throughput-limiting component in processing flow. Typically, both the predecessor and successor components of a bottleneck are capable of providing higher aggregate throughput than that of a bottleneck. Bottlenecks may sometimes be

replaced by a less limiting implementation (optimized); this may cause a dominant program bottleneck to move to another location in the code. Note that not every hotspot is necessarily a bottleneck. Many tightly optimized numeric libraries, for example, will spend nearly all of their time performing FPU computations, but this does not mean they are inefficient (the evidence for this is provided when the machine reaches performance near its hardware peak or close to the theoretical throughput limit of the computational algorithm used). Profilers may record compute performance data at the system level (including all active processes, system daemons, and kernel code running on a node), the process level, where only data relevant to a specific process is collected, or at the level of individual threads of a process. Additionally, profiling may be restricted to a user space, a kernel space, or both. Profiling requires that the analyzed application is *instrumented*, or modified in a way that permits the profiler to access the required runtime information. This process may be more invasive (e.g., the programmer injecting the required function calls or macros in the relevant places of source code) or less so (linking with a profiling library or attaching an external profiler to an already running process). The former often occurs when the tracking of user-defined events is necessary.

Besides analysis of computational performance, profiling tools may monitor other characteristics of the executed programs. One is memory usage over the course of program execution. This may apply to the overall size of virtual memory allocated by the application, the amount of physical memory assigned to the program, the shared memory that may be accessible to other concurrently executing processes, and the sizes of the program's stack, data, and text segments. The other aspect is I/O, for which the profiler may record the number of I/O operations, the amount of data transferred to or from the secondary storage or buffer cache, achieved data bandwidth, number of files opened, and so on. Finally, communication profiling registers the number and size of messages sent, their destinations, latencies, and bandwidths. This can be further categorized by network type (Ethernet, InfiniBand, etc.), communication endpoint type (sockets, RDMA), or protocol used. Information collected during profiling may be used to classify a program or its individual subroutines as *CPU* (or *compute*) *bound*, where execution time is dominated by processor speed, *memory bound*, for which execution time is primarily dictated by the amount of memory needed to store the program's data structures, or *I/O bound*, where a dominant fraction of execution time is spent performing I/O operations. It is worth noting that the code characteristic may change as a result of optimization, e.g., from CPU bound to I/O bound.

13.3.2 ESSENTIAL *GPERFTOOLS*

The *time* utility discussed previously is an example of a simple profiler. Its usefulness is limited by reporting only the single average, cumulative, or maximum value of parameters of interest for the entire duration of program execution. This makes it impossible to pinpoint the moment in program execution when performance was degraded and cross-reference it to the responsible sections of source code.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 410

Modern profiling tools attempt to address this issue. One of the commonly used profilers is available as a part of the *gperftools* [7] package. While originally named Google Performance Tools, the code is currently maintained by the community and distributed under the BSD license. It provides a statistical CPU profiler, *pprof*, and several tools based around the *tcmalloc* (thread-caching malloc) library. Besides offering an improved memory allocation library for multithreaded environments, *tcmalloc* library supports memory leak detection and dynamic memory allocation profiling. To illustrate the use of these features, the program from Code 13.1 was compiled using the command shown below (note the addition of `-lprofiler` to the command line). To permit access to the program's symbol table, a `-ggdb` option was specified as well:

```
> gcc -O2 -ggdb mvmult.c -o mvmult -lcblas -lprofiler
```

The *gperftools* CPU profiler does not require any changes to the source code, and after successfully linking the instrumented application may be executed. The location of the file containing the collected data must be specified using the CPUPROFILE environment variable, as demonstrated below:

```
> env CPUPROFILE=mvmult.prof ./mvmult 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)
PROFILE: interrupts/evictions/bytes = 115/0/376
```

The program execution proceeds as before, with the expected output appearing on the console. The only change is the final line, which confirms that the profiling indeed took place and collected 115 data samples. To display the obtained information, the `pprof` tool is used:

```
> pprof --text mvmult mvmult.prof
Using local file mvmult.
Using local file mvmult.prof.
Total: 115 samples
      58  50.4%  50.4%      58 50.4% ddot_
      57  49.6% 100.0%      57 49.6% init
       0   0.0% 100.0%      57 49.6% 0x00007f2c9485e00f
```

The produced output is organized in several columns. The first shows the sample count associated with each function. Whenever the profiler collects a sample, it records, among other things, the current address stored in the instruction pointer of the running program context. Subsequent analysis done by `pprof` assigns the collected addresses to individual program functions. This is shown

in the second column. The result above indicates that practically the entire program time is spent in two functions, `ddot_` and `init`. While `init` may be found in Code 13.1, `ddot_` is a Fortran function indirectly called by CBLAS that computes the double-precision dot product. The third column lists the cumulative percentage of samples for all functions displayed so far. The fourth and fifth columns deal with the aggregate sample counts and percentages for the annotated function as well as all its callees. Hence the unnamed function in the last line is the likely ancestor of the `init` function; it might be related to an early setup code that executes before the invocation of `main`. Finally, the last column lists the affected function names, or if not available, the raw sampled addresses.

The default sampling frequency is 100 samples per second. This can be set to a custom value using the `CPUPROFILE_FREQUENCY` environment variable, although the maximum speed for most Linux platforms is limited to 1000 per second. Since the test application runs for only about a second, trying to increase the number of samples may offer additional insights:

```
> env CPUPROFILE=mvmult1K.prof CPUPROFILE_FREQUENCY=1000 ./mvmult 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)
PROFILE: interrupts/evictions/bytes = 1147/0/536
```

About 10 times as many samples were collected. Their analysis reveals the following:

```
> pprof --text mvmult mvmult1K.prof
Using local file mvmult.
Using local file mvmult1K.prof.
Total: 1147 samples
     576  50.2%  50.2%      576  50.2% ddot_
     571  49.8% 100.0%      571  49.8% init
       0   0.0% 100.0%      571  49.8% 0x00007f5fd0cda00f
```

It is apparent that most of the test application execution is indeed concentrated in the two functions identified before. However, `pprof` supports other analysis options that may be changes through command line switches:

- `--text` displays the profile in a plain-text form
- `--list=<regex>` outputs only data related to functions whose names match the provided regular expression
- `--disasm=<regex>` is like list, but performs disassembly of relevant section of the program while annotating each line with a sample count

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 412

- `--dot`, `--pdf`, `--ps`, `--gif`, and `--gv` generate annotated graphical representation of a call graph and output it to *stdout* in the requested format. Requires that the *dot* converter is installed in the system. The last option uses *gv* viewer to open a window with call graph visualization.

The default output of `pprof` is performed at function granularity, but sometimes it is useful to change this to avoid lengthy output or zoom in more closely on to the source of the problem. Adjustment options, in order of decreasing resolution, are:

- `--addresses` shows annotated code addresses
- `--lines` annotates source code lines
- `--functions` lists the statistics per function
- `--files` switches to whole-file granularity.

To see how the samples are distributed within the `init` function, one may apply the following command to the set of profiling data collected before (to save space, the produced output was truncated and removed lines were replaced with "`[...]`"):

```
> pprof --list=init --lines mvmult mvmult1K.prof
Using local file mvmult.
Using local file mvmult1K.prof.
ROUTINE ==================== init in /home/maciek/perf/mvmult.c
   571     571 Total samples (flat / cumulative)
[...]
     .       .    6: void init(int n, double **m, double **v, double **p,
int trans) {
     .       .    7:   *m = calloc(n*n, sizeof(double));
     .       .    8:   *v = calloc(n, sizeof(double));
     .       .    9:   *p = calloc(n, sizeof(double));
     .       .   10:   for (int i = 0; i < n; i++) {
     1       1   11:     (*v)[i] = (i & 1)? -1.0: 1.0;
     .       .   12:     if (trans) for (int j = 0; j <= i; j++) (*m)[j*n+i]
= 1.0;
   570     570   13:     else for (int j = 0; j <= i; j++) (*m)[i*n+j] =
1.0;
     .       .   14:   }
     .       .   15: }
[...]
```

Not unexpectedly, this shows that most initialization time is spent within the main loop. Of that, the inner loop performing initialization of matrix rows dominates the execution time, while the multiplicand vector initialization is marginal by comparison. Since the sources of BLAS routines are not

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 413

available, a disassembled code listing may be used to identify the fine-grain hotspots in that code (again, the output was shortened to the most interesting fragment):

```
> pprof --disasm=ddot_ mvmult mvmult.prof
Using local file mvmult.
Using local file mvmult.prof.
ROUTINE ====================== ddot_
   576    576 samples (flat, cumulative) 50.2% of total
[...]
    48     48          fcc0: movsd  -0x8(%rax),%xmm0
     9      9          fcc5: add    $0x28,%rax
     .      .          fcc9: add    $0x28,%rcx
    60     60          fccd: movsd  -0x20(%rax),%xmm2
    43     43          fcd2: mulsd  -0x30(%rcx),%xmm0
     .      .          fcd7: mulsd  -0x20(%rcx),%xmm2
     2      2          fcdc: addsd  %xmm0,%xmm1
    26     26          fce0: movsd  -0x28(%rax),%xmm0
     .      .          fce5: mulsd  -0x28(%rcx),%xmm0
     .      .          fcea: addsd  %xmm0,%xmm1
    81     81          fcee: addsd  %xmm2,%xmm1
    93     93          fcf2: movsd  -0x18(%rax),%xmm2
     9      9          fcf7: mulsd  -0x18(%rcx),%xmm2
     .      .          fcfc: movapd %xmm1,%xmm0
    57     57          fd00: movsd  -0x10(%rax),%xmm1
    13     13          fd05: mulsd  -0x10(%rcx),%xmm1
     .      .          fd0a: cmp    %rax,%rdx
     .      .          fd0d: addsd  %xmm2,%xmm0
    70     70          fd11: addsd  %xmm0,%xmm1
    65     65          fd15: jne    fcc0 <ddot_+0x110>
[...]
```

It is not difficult to guess that the annotated instructions are performing the arithmetic operations (scalar double-precision multiplication and addition) and managing the data movement between memory and floating-point registers (here denoted as %xmm with a numeric suffix). The listed code segment captures the innermost loop, as evidenced by the backward conditional branch in the last line. The overhead of memory access is comparable to the cost of computation. The fact that only scalar arithmetic operations were used indicates an optimization opportunity, since the dense algebra algorithms frequently benefit from SIMD support available on modern CPUs. Further investigation reveals that CBLAS was linked to the reference BLAS library rather than to of any of the optimized versions.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 414

For completeness, profile data of the transposed case sampled at 100 samples/second is available below. While the program's execution is still confined to the same functions as before, the ratio of data timing has changed: initialization is less affected by column-major layout. At this point it is difficult to ascertain the reason for the difference in performance based solely on CPU profile data.

```
> pprof --text mvmult mvmult_trans.prof
Using local file mvmult.
Using local file mvmult_trans.prof.
Total: 13577 samples
    9240  68.1%  68.1%     9240  68.1% ddot_
    4335  31.9% 100.0%     4335  31.9% init
       2   0.0% 100.0%        2   0.0% ddotsub_
       0   0.0% 100.0%     4335  31.9% 0x00007f6440b6900f
```

One of *gperftools* features is the ability to detect memory leaks. To enable this functionality, it is necessary to link the application with the *tcmalloc* library or set the environment variable `LD_PRELOAD` to `libtcmalloc.so`. Before launching the application, the leak detector needs to be informed about the flavor of checking that should be performed. This is accomplished by storing one of the keywords (`minimal`, `normal`, `strict`, or `draconian`) in the `HEAPCHECK` environment variable. They differ in scope and level of detail performed by the heap allocation checker; for most purposes `normal` mode is sufficient. The compilation command line and results of the instrumented program execution are shown below.

```
> gcc -O2 mvmult.c -o mvmult -lcblas -ltcmalloc
> env HEAPCHECK=normal ./mvmult 20000
WARNING: Perftools heap leak checker is active -- Performance may suffer
tcmalloc: large alloc 3200000000 bytes == 0xe9e000 @ 0x7f887688eae7
0x4009b1 0x400b95
Size 20000; abs. sum: 10000.000000 (expected: 10000)
Have memory regions w/o callers: might report false leaks
Leak check _main_ detected leaks of 3200160000 bytes in 2 objects
```

Since the program in Code 13.1 performs explicit memory allocation in `init` and that memory is never freed, the heap checker reports a leak at the end of `main`. Note that *tcmalloc* prints statements whenever large amounts of memory are allocated.

The tool may also profile memory management, similarly to CPU profiling. In this case the source code needs to be explicitly instrumented: a `HeapProfilerStart` function has to be inserted before the profiled section of code, and a `HeapProfilerStop` function must be added at the end. The first function takes one argument describing the file name prefix used to store the profiling data (since multiple files may be generated, each has a unique number and ".prof" extension added automatically). The

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 415

prototypes of these functions are defined in the header file "gperftools/heap-profiler.h". The profiler's behavior may be adjusted through dedicated environment variables, detailed below.

- HEAP_PROFILE_ALLOCATION_INTERVAL: each time the specified number of bytes is allocated the profile data is stored in file. Allocation interval defaults to 1 GB.
- HEAP_PROFILE_INUSE_INTERVAL: as above, but the profile is written every time the total memory use by the program increases by the specified value, defaulting to 100 MB.
- HEAP_PROFILE_TIME_INTERVAL: stores data for every time period in seconds (default: 0).
- HEAP_PROFILE_MMAP: in addition to the usual C and C++ memory allocation calls such as `malloc`, `calloc`, `realloc`, and `new`, `this` also profiles `mmap`, `mremap`, and `sbrk` calls. By default it is disabled (false).
- HEAP_PROFILE_ONLY_MMAP: constraints the profiling to only `mmap`, `mremap`, and `sbrk` functions; the default value is false.
- HEAP_PROFILE_MMAP_LOG: enables logging of `mmap` and `munmap` calls; default is false.

To illustrate the use of the memory profiler, the following sequence of commands compiles the instrumented application (the file prefix was set to "mvmult") and launches it with profiling enabled. The threshold is set to a low value to capture all allocation calls.

```
> env HEAP_PROFILE_ALLOCATION_INTERVAL=1 ./mvmult_heap 20000
Starting tracking the heap
tcmalloc: large alloc 3200000000 bytes == 0x2258000 @ 0x7fd915a2eae7
0x400a71 0x400c55
Dumping heap profile to mvmult.0001.heap
(3051 MB allocated cumulatively, 3051 MB currently in use)
Dumping heap profile to mvmult.0002.heap
(3051 MB allocated cumulatively, 3051 MB currently in use)
Dumping heap profile to mvmult.0003.heap
(3052 MB allocated cumulatively, 3052 MB currently in use)
Dumping heap profile to mvmult.0004.heap
(3052 MB allocated cumulatively, 3052 MB currently in use)
Size 20000; abs. sum: 10000.000000 (expected: 10000)
```

After the program execution completes, four data dump files may be found in working directories named from "mvmult.0001.heap" to "mvmult.0004.heap". The pprof may display the information in one of four modes determined by the additional command-line switch:

- `--inuse-space`:—shows the number of megabytes currently in use (the default)
- `--inuse-objects`—shows the number of objects in use
- `--alloc_space`—shows the number of allocated megabytes
- `--alloc-objects`—shows the number of allocated objects.

Thus to display the allocated data captured by the last sample, the following command is used:

```
> pprof --text --alloc_space mvmult_heap mvmult.0004.heap
Using local file mvmult_heap.
Using local file mvmult.0004.heap.
Total: 3052.1 MB
  3052.1 100.0% 100.0%   3052.1  100.0% init
     0.0   0.0% 100.0%      0.0    0.0% __GI__IO_file_doallocate
     0.0   0.0% 100.0%      0.2    0.0% 0x00000000c0e19fff
     0.0   0.0% 100.0%   3051.9  100.0% __libc_csu_init
```

While the *gperftool* suite directly supports profiling of individual applications, it is also possible to use it for inspection of MPI programs. Since application performance data must be written to a specific file, one way to avoid collisions is to make sure that each monitored MPI process is assigned a different file. This is accomplished by adding the following statement to the application's source at a point following `MPI_Init` invocation:

`ProfilerStart(`*filename*`);`

The prototype of this function is available in `gperftools/profiler.h` along with other calls that may be helpful to control the profiler's operation. The *filename* parameter must be a different string for each MPI process. This is typically arranged by deriving it from the rank of the process within `MPI_COMM_WORLD`. For example:

```
int rank;
MPI_Comm_rank(MPICOMM_WORLD, &rank);
char filename[256];
snprintf(filename, 256, "my_app%04d.prof", rank);
ProfileStart(filename).
```

13.4 MONITORING HARDWARE EVENTS

13.4.1 PERF

The *perf* framework [8], also referred to as *perf_events*, is a performance monitoring tool and event tracer closely integrated with the Linux OS kernel. Its primary functionality is based on the *sys_perf_event_open* [9] system call introduced in the 2.6 series of Linux. The system call enables access to special-purpose registers of the CPU that may be configured to collect the counts of specific hardware-level events. These events may vary from processor to processor, but their main categories include the following:

- Cache related: misses and references issued. These may be further grouped by cache level (L1 through L3), cache type (instruction and data), and access type (loads and stores).

- Translation lookaside buffer related. These may also be subdivided into instruction and data categories, and by access type (load/store).
- Branch statistics. These include counts of overall branch occurrences and missed branch target loads.
- Instructions and cycles. *Perf* can provide the number of executed instructions or the count of CPU cycles that occurred during program execution.
- Stalled or idle cycles. These further subdivide into front-end and back-end stalls. The first indicates inability to fill completely the available capacity of the first stages of the execution pipeline, and may be caused by instruction cache or translation lookaside buffer (TLB) misses, mispredicted branches, or unavailability of translation into microoperations for specific instruction(s). The back-end issues may be caused by interinstruction dependencies (e.g., a long-latency instruction delaying the execution of other dependent instructions, such as division) or availability of memory units.
- Node-level statistics: prefetches, loads and stores, and misses. Prefetch misses are counted separately to avoid false inflation of statistics describing actual data accesses generated by the monitored code.
- Data collected by the processor's performance management unit (PMU). These counters provide the aggregate values for the whole CPU, including primarily *uncore*-related events. Uncore is a term coined by Intel to describe segments of CPU logic that are not parts of the core execution pipeline and thus are shared by the cores. They include memory controllers and their interfaces, a node-level interconnect bus that provides NUMA functionality, last-level cache, a coherency traffic monitor, and power management.

The *perf* tool also provides access to many software-level kernel events that may be of great use for performance analysis. They comprise counts of context switches, context migrations, data alignment faults, major, minor, and aggregate page faults, accurate time measurements, and custom events defined using the Berkeley Packet Filter framework. The complete list of events supported on the local system is obtained with:

```
> perf list
```

Perf may be invoked in several modes of operation selected by the first argument on the command line. The frequently used commands are:

- `stat`, which executes the provided application with arguments while collecting the counts of specified events or a default event set
- `record`, which enables per thread, per process, or per CPU profiling
- `report`, which performs analysis of data collected by `records`
- `annotate`, which correlates the gathered profiling data to assembly code
- `top`, which displays the statistics in real time using format resembling that of the Unix *top* utility for visualization of process activity
- `bench`, which invokes a number of predefined kernel benchmarks.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 418

To test this functionality in practice, we can profile the test application shown in Code 13.1. The result for row-major (nontransposed) mode is presented below.

```
> perf stat ./mvmult 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)

 Performance counter stats for './mvmult 20000':

       1219.404556     task-clock (msec)         #    1.000 CPUs utilized
                 1     context-switches          #    0.001 K/sec
                 0     cpu-migrations            #    0.000 K/sec
           781,490     page-faults               #    0.641 M/sec
     3,898,266,727     cycles                    #    3.197 GHz
     2,283,166,328     stalled-cycles-frontend   #   58.57% frontend
cycles idle
     1,372,252,385     stalled-cycles-backend    #   35.20% backend
cycles idle
     3,764,331,355     instructions              #    0.97  insns per
cycle
                                                 #    0.61  stalled cycles
per insn
       495,220,268     branches                  #  406.116 M/sec
           815,338     branch-misses             #    0.16% of all
branches

       1.219967824 seconds time elapsed
```

Invoking the same for a column-major layout produces the following.

```
Performance counter stats for './mvmult 20000 t':

      12212.530334     task-clock (msec)         #    1.000 CPUs utilized
                11     context-switches          #    0.001 K/sec
                 0     cpu-migrations            #    0.000 K/sec
         1,213,417     page-faults               #    0.099 M/sec
    42,933,883,759     cycles                    #    3.516 GHz
    39,567,001,587     stalled-cycles-frontend   #   92.16% frontend
cycles idle
    37,181,761,140     stalled-cycles-backend    #   86.60% backend
cycles idle
     6,077,067,370     instructions              #    0.14  insns per
cycle
                                                 #    6.51  stalled cycles
per insn
       918,790,187     branches                  #   75.233 M/sec
         1,276,503     branch-misses             #    0.14% of all
branches

      12.213751102 seconds time elapsed
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 419

Besides the duration of program execution, there are several other noticeable differences between the two modes of operation. Firstly, the number of front-end and back-end stalls is significantly increased. The effective number of stalls per instruction is an order of magnitude higher. The instruction throughput per cycle is also much lower. This suggests that serious inefficiencies are introduced in the processing pipeline. Curiously, despite using nearly identical algorithms, the number of executed instructions is 60% greater for the column-major case. The code also encounters a much higher number of page faults in that mode.

Since the types of executed instructions are likely similar for both cases, the increased number of stalls may indicate caching issues. The higher count of page faults might also suggest TLB problems. To confirm this, the codes are reexecuted with custom selection of events. Note that *perf* may accommodate a greater number of events in a single invocation than available hardware slots in the processor using a technique called *multiplexing*. It means that at any given moment only a subset of requested events is configured on the processor; this subset is periodically replaced with one that contains other requested events. This is repeated cyclically to permit all specified events to be active for an approximately equal share of time during application execution. The additional options that may be passed to *perf* invocation are listed below.

- `-e` *event* [:*modifier*][,*event*[:*modifier*]]...

  Explicitly specifies the kinds of monitored events. Each event name may be followed by one or more modifiers, such as `u` for measuring only the events when the application executes in user mode or `k` when it is in kernel mode (and others which are not relevant here).

- `-B`

  Separates groups of every three digits in numbers by commas for easier readability.

- `-p` *pid*

  Instead of directly launching an application, the profiler attaches to an existing process with the specified *pid*.

- `-r` *integer*

  Repeatedly runs the command, collecting the aggregate statistics. The result shows the mean values for each event and deviation from the mean.

- `-a`

  Forces *perf* to collect data for all CPUs, including profiles of other applications running at the same time. The default is to monitor only the specified application's threads.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 420

To put this into practice, the code was run again with monitoring of cache misses and TLB load misses enabled:

```
> perf stat -B -e cache-misses,dTLB-load-misses,iTLB-load-misses ./mvmult
20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)

 Performance counter stats for './mvmult 20000':

        29,307,244      cache-misses
         3,121,156      dTLB-load-misses
             4,224      iTLB-load-misses

       1.227144489 seconds time elapsed
```

And in transposed version:

```
Performance counter stats for './mvmult 20000 t':

        79,004,606      cache-misses
       405,044,765      dTLB-load-misses
            33,124      iTLB-load-misses

      12.185000849 seconds time elapsed
```

The collected data shows a significant increase for all three figures. Particularly damaging is the two orders of magnitude jump in data-TLB misses. This is caused by strided access to matrix elements; the consecutive references not only touch different cache lines but involve different memory pages (eight-byte entries with 20,000 element stride are effectively separated by 160 KB, which is far greater than the default page size of 4 KB). This emphasizes the importance of selecting algorithms that exhibit good spatial locality of access.

To verify that the change is caused by different data layouts used by the main compute functions, the performance data was recorded in sampling mode using the command shown below. The `-F` option controls the sampling frequency; in this case 1000 samples per second are requested.

```
> perf record -F 1000 -e cache-misses,dTLB-load-misses,iTLB-load-misses
./mvmult 20000 t
Size 20000; abs. sum: 10000.000000 (expected: 10000)
[ perf record: Woken up four times to write data ]
[ perf record: Captured and wrote 0.834 MB perf.data (17,967 samples) ]
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 421

The collected information may be analyzed using the "`perf report`" command. The most significant excerpts of the result are listed below.

```
# Samples: 6K of event 'cache-misses'
# Event count (approx.): 78141963
#
# Overhead  Command  Shared  Object     Symbol
# ........  .......  ......  ......     ................................
#
    33.64%  mvmult   libblas.so.3.6.0   [.] ddot_
    27.12%  mvmult   [kernel.vmlinux]   [k] clear_page
    24.04%  mvmult   mvmult             [.] init
     6.73%  mvmult   [kernel.vmlinux]   [k] _raw_spin_lock
     3.93%  mvmult   [kernel.vmlinux]   [k] page_fault
[...]
# Samples: 10K of event 'dTLB-load-misses'
# Event count (approx.): 405199968
#
# Overhead  Command  Shared  Object     Symbol
# ........  .......  ......  ......     ....................................
#
    99.03%  mvmult   libblas.so.3.6.0   [.] ddot_
     0.63%  mvmult   [kernel.vmlinux]   [k] page_fault
     0.14%  mvmult   [kernel.vmlinux]   [k] handle_mm_fault
[...]
# Samples: 1K of event 'iTLB-load-misses'
# Event count (approx.): 33857
#
# Overhead  Command  Shared  Object     Symbol
# ........  .......  ......  ...........
......................................
#
    15.57%  mvmult   libblas.so.3.6.0   [.] ddot_
     8.86%  mvmult   libcblas.so        [.] cblas_ddot
     6.16%  mvmult   mvmult             [.] init
     5.97%  mvmult   [kernel.vmlinux]   [k] cpumask_any_but
     5.74%  mvmult   [kernel.vmlinux]   [k] page_fault
     5.54%  mvmult   [kernel.vmlinux]   [k] notifier_call_chain
     4.62%  mvmult   [kernel.vmlinux]   [k] flush_tlb_mm_range
     4.57%  mvmult   libcblas.so        [.] ddotsub_
     3.27%  mvmult   [kernel.vmlinux]   [k] smp_apic_timer_interrupt
     2.90%  mvmult   [kernel.vmlinux]   [k] apic_timer_interrupt
     2.33%  mvmult   [kernel.vmlinux]   [k] update_vsyscall
     2.10%  mvmult   mvmult             [.] mult
[...]
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 422

As can be seen, the ddot_ function is the primary contributor of cache and TLB misses. A large percentage of cache misses are also caused by the kernel's page-clearing function, most likely called as a consequence of using the calloc function to allocate the memory for matrix and vectors. Not surprisingly, the init function is the source of a significant fraction of cache misses.

Unlike *gperftools*, *perf* can only record the performance data in a file with a fixed name. This makes it harder to analyze the performance of all component processes comprising an MPI application. The workaround on a machine with dedicated local storage partitions (e.g., in /tmp) could be by starting the application in node-exclusive mode (one process per node) after changing the working directory to one on the local file system. After the application terminates, the generated data files may be copied (and renamed) for analysis to a shared file system using *scp*. If all component processes of the application execute a similar workload, it may suffice to set up monitoring for only one of them, as described in Section 3.5.2.2. The approximate counts for the whole application are then derived by multiplying the single process count by the number of executed MPI processes. Note that monitoring of arbitrary rank can also be arranged by subdividing the processes into correctly sized groups using the -np *n* option to *mpirun*, while remembering that they have to add up to the total count of processes required by the application and only one instance may invoke *perf*.

13.4.2 PERFORMANCE APPLICATION PROGRAMMING INTERFACE

The Performance Application Programming Interface (PAPI) [10] is a performance monitoring toolkit developed at the University of Tennessee Innovative Computing Laboratory. It provides C and Fortran library and header files containing prototypes of functions that may be used to instrument user applications. The application programming interface (API) categories comprise library initialization and shutdown, event description and translation between symbolic event names and their codes, creation and manipulation of event sets, starting and stopping of event counters, retrieval, accumulation, resetting, and initialization of counter values, system parameter queries, and various timing functions. The package also provides a number of practical utilities.

- papi_avail prints the symbolic names of *preset* events annotated with availability flags on the local systems and noting whether they are counted directly or derived by using more than one counter. Using option -a limits the display only to events locally available.
- papi_native_avail similarly displays so-called *native* events, which typically comprise uncore and node-level events.
- papi_decode outputs more detailed event descriptions in comma-separated values (csv) format.
- papi_clockres determines the practical resolution of various time and cycle measurement interfaces.
- papi_cost checks the latency of invocation of various API functions in different configurations.
- papi_event_chooser prints out events that may be added without conflict to a set containing events specified by the user.
- papi_mem_info shows the local machine cache and TLB hierarchy information.

PAPI events are less portable across processor architectures than those exposed by the *perf* tool. The user always needs to confirm whether a specific event is available on the target platform by using `papi_avail` or `papi_native_avail`. Due to the growing complexity of microprocessor designs, the interpretation of seemingly the same events may change even between different iterations of the same architecture. On the other hand, PAPI may be used to instrument the application in precise locations of the code and enables use of events that are normally not supported by *perf*.

To showcase the use of the interface, Code 13.1 was instrumented with two counters that tally the occurrences of double-precision operations, but one counts instances of all floating-point operations converted to scalar operations (`PAPI_DP_OPS`) and the other counts all vector operations (`PAPI_VEC_DP`). The counters are activated just before initialization (line 45) and their values are retrieved after return from the `init`, `mult`, and `cblas_dasum` functions (lines 48, 50, and 52). To guard against silent failures, a PAPI_CALL macro was defined in lines 25–30 to verify that called PAPI routines are completed successfully. As before, only the modified portion of the source code is provided (not counting the inclusion of the PAPI header, `papi.h`, in the top section of the source file) in Code 13.3.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cblas.h>
4  #include <time.h>
5  #include <papi.h>
6
7  void init(int n, double **m, double **v, double **p, int trans) {
8    *m = calloc(n*n, sizeof(double));
9    *v = calloc(n, sizeof(double));
10   *p = calloc(n, sizeof(double));
11   for (int i = 0; i < n; i++) {
12       (*v)[i] = (i & 1)? -1.0: 1.0;
13       if (trans) for (int j = 0; j <= i; j++) (*m)[j*n+i] = 1.0;
14       else for (int j = 0; j <= i; j++) (*m)[i*n+j] = 1.0;
15   }
16 }
17
18 void mult(int size, double *m, double *v, double *p, int trans) {
19 int stride = trans? size: 1;
20   for (int i = 0; i < size; i++) {
21     int mi = trans? i: i*size;
22     p[i] = cblas_ddot(size, m+mi, stride, v, 1);
23 } }
24
25 #define PAPI_CALL(fn, ok_code) do { \
26   if (ok_code != fn) { \
27     fprintf(stderr, "Error: " #fn " failed, aborting\n"); \
28     exit(1); \
29   } \
30 } while (0)
31
```

```
32 #define NEV 2
33
34 int main(int argc, char **argv) {
35   int n = 1000, trans = 0;
36   if (argc > 1) n = strtol(argv[1], NULL, 10);
37   if (argc > 2) trans = (argv[2][0] == 't');
38
39   int evset = PAPI_NULL;
40   PAPI_CALL(PAPI_library_init(PAPI_VER_CURRENT), PAPI_VER_CURRENT);
41   PAPI_CALL(PAPI_create_eventset(&evset), PAPI_OK);
42   PAPI_CALL(PAPI_add_event(evset, PAPI_DP_OPS), PAPI_OK);
43   PAPI_CALL(PAPI_add_event(evset, PAPI_VEC_DP), PAPI_OK);
44   double *m, *v, *p;
45   PAPI_CALL(PAPI_start(evset), PAPI_OK);
46   init(n, &m, &v, &p, trans);
47   long long v1[NEV], v2[NEV], v3[NEV];
48   PAPI_CALL(PAPI_read(evset, v1), PAPI_OK);
49   mult(n, m, v, p, trans);
50   PAPI_CALL(PAPI_read(evset, v2), PAPI_OK);
51   double s = cblas_dasum(n, p, 1);
52   PAPI_CALL(PAPI_stop(evset, v3), PAPI_OK);
53   printf("Size %d; abs. sum: %f (expected: %d)\n", n, s, (n+1)/2);
54   printf("PAPI counts:\n");
55   printf(" init: event1: %15lld event2: %15lld\n", v1[0], v1[1]);
56   printf(" mult: event1: %15lld event2: %15lld\n", v2[0]-v1[0], v2[1]-v1[1]);
57   printf(" sum: event1: %15lld event2: %15lld\n", v3[0]-v2[0], v3[1]-v2[1]);
58   return 0;
59 }
```

Code 13.3. Instrumented section of Code 13.1 for collection of floating-point operation counts using PAPI.

For correct compilation, the program must be linked with the PAPI library, as shown below.

```
> gcc -O2 mvmult_papi.c -o mvmult_papi -lcblas -lpapi.
```

Running the instrumented program produces the following output.

```
> ./mvmult_papi 20000
Size 20000; abs. sum: 10000.000000 (expected: 10000)
PAPI counts:
  init: event1:          0  event2:     0
  mult: event1:  804193640  event2:     0
   sum: event1:      20276  event2:     0
```

Since the reference BLAS implementation does not use vector floating points, the count of vector operations stays at zero. The theoretical count of scalar operations should be $20{,}000^2$ for multiplication and 20,000∗19,999 for addition, for a grand total of 799,980,000 in the `mult` function and 19,999 operations (since the absolute value computation requires only clearing the sign bit) in `cblas_dasum`. The counters consistently register higher values, most likely due to speculative execution.

After replacing the reference BLAS library by a highly optimized Intel Math Kernel Library [11] that takes advantage of vector instructions supported by the target machine, the application was reexecuted to produce the following values.

```
PAPI counts:
 init: event1:          0  event2:         0
 mult: event1: 1055372246  event2: 527686123
  sum: event1:      24674  event2:     12337
```

The count of vector operations was roughly half of the scalar figure. This indicates that the library selected the use of vector instructions with two floating-point numbers per instruction. Indeed, the machine on which the test was performed supports Streaming SIMD Extensions (SSE) instruction set with up to two operands per vector. As a result of this change, the execution time dropped from 1.22 s to 1.08 s.

13.5 INTEGRATED PERFORMANCE MONITORING TOOLKITS

Software application components do not operate independently: not only do they have to share various system resources, such as processor cores, memory, storage, and network bandwidth, but they must also coexist with the periodically executing operating system threads, service daemons, and other applications. While the last issue is mitigated to some extent by properly configured job managers that schedule new processes on shared resources only when permitted by the owner of the already executing job on the node, the resultant application performance is the outcome of multiple factors, frequently acting against each other. To gain more complete understanding of an application's behavior, it therefore makes sense to create performance monitors that combine various aspects of application profiling in a single package that permits easy visualization and comparison of performance data.

One such tool is the Tuning and Analysis Toolkit (TAU) [12] developed at the Performance Research Laboratory at the University of Oregon and distributed under the BSD license. TAU may be used in single-node and distributed environments, including 32-bit and 64-bit Linux clusters, ARM platforms, Windows machines, Cray computers running Compute Node Linux, IBM BlueGene and POWER families on AIX and Linux, NEC SX series, and AMD, Nvidia, and Intel GPUs as well as a number of older architectures. In addition to instrumentation (for profiling or tracing), measurement, analysis, and visualization, it is capable of managing performance information databases and performing data mining functions. For graphical display of collected data TAU provides a Java-based *paraprof* visualization tool. Supported languages include C, C++, Fortran, UPC, Python, Java, and Chapel.

Event types recognized and captured by TAU include *interval* and *atomic* events. Interval events have defined start and end points. The statistics derived from interval event measurement may be *inclusive*, where outer intervals include event counts or timing collected for all nested intervals, or *exclusive*, when the resultant data shows only values for event counts or times that are relevant solely to the specified interval but excludes the statistics for all its "children" intervals. Interval metrics are monotonic—they may only increase during program execution (e.g., when a monitored function is reinvoked). Atomic events capture momentary metric values related to computation state at predefined trigger points. They may vary throughout the execution of the application. TAU captures them as a total (cumulative) value, minimum, maximum, average, and number of samples collected. The user-defined events may be of both interval and atomic kinds. In addition, execution context may be attached to atomic events to determine the calling path taken.

TAU supports three instrumentation methods that differ in level of their provided features.

- **Source-level** instrumentation is the most flexible method. This is the only mode supporting insertion of user-defined probes. Using this method permits exclusion of regions of code that are not critical for program performance or otherwise not interesting from the output. It also allows profiling of various low-level events, such as loops or program phases. This is accomplished by static analysis of source code using the Program Database Toolkit (PDT) package, creating a modified copy of sources, and compilation of the instrumented code.
- **Library-level** instrumentation is employed in cases when sources are not available, for example when monitoring of external or system libraries is needed. It applies wrapper libraries that may be used with static or dynamic libraries under investigation. In both cases symbol rewriting techniques are used (such as weak symbols for static libraries and library preloading for dynamic libraries) that redefine functions associated with specific identifiers, thus permitting interception of user calls and insertion of appropriate monitoring code.
- **Binary code** instrumentation requires *Dyninst* [13], developed by the Paradyn Tools Project. While the least invasive of all the described methods, it does not support many features available using other instrumentation approaches. Binary instrumentation is performed by rewriting binary application code, hence it may be used with already linked applications and without requiring any access to source files.

To demonstrate necessarily only a very few options from TAU's broad palette of supported configurations and measurements, Code 13.1 has been transformed using PDT-driven source instrumentation and compiled using the following command:

```
> taucc -tau:verbose -tau:pdtinst -optTauSelectFile=select.tau mvmult.c -O2
-o mvmult -lcblas -lm
```

While not required by the application, math library (-lm) had to be added to the command line to avoid linker complaints. TAU installation may support several different configurations involving on occasion options that may not be specified at the same time. The conflicts are avoided by encoding such configurations into separate Makefiles with names suffixed with the applicable configuration

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 427

options. To point the TAU compiler toward the most relevant option, a suitable environment variable needs to be set:

```
> export TAU_MAKEFILE=/opt/tau/x86_64/lib/Makefile.tau-memory-phase-papi
-mpi-pthread-pdt
```

Of course, the installation path and the actual Makefile name have to be modified as appropriate on the local host.

The compilation command presented above illustrates simple selective instrumentation defined in the file `select.tau`. Its content is as follows.

```
BEGIN_EXCLUDE_LIST
void cblas_dasum(int, double *, int)
END_EXCLUDE_LIST

BEGIN_FILE_EXCLUDE_LIST
*.so

END_FILE_EXCLUDE_LIST
BEGIN_INSTRUMENT_SECTION
loops file="mvmult.c" routine="mult"
memory file="mvmult.c" routine="init"
END_INSTRUMENT_SECTION
```

This instructs TAU to exclude `cblas_dasum` (which earlier measurements showed to be nonessential to program performance) from profiling, as well as all dynamic libraries (since they contain system-level CBLAS and BLAS routines that are not the subject of investigation). TAU is also supposed to provide loop-level instrumentation in the `mult` function and memory instrumentation in `init`. Note that the wildcard character for function specification is "#" to avoid confusion with pointer syntax.

To collect data during the application's execution, TAU needs additional guidance on whether to profile or trace the application, what execution parameters to capture, and what type of hardware events should be collected. This is accomplished via environment variables, e.g.:

```
> TAU_METRICS=TIME
> TAU_PROFILE=1
```

The TAU_METRICS variable may contains several metric identifiers, including preset and native PAPI event names, separated by colons. After execution of the instrumented program is complete, a number of `profile.x.y.z` files, where x, y, and z are numbers corresponding to nodes (MPI ranks), contexts, and thread numbers, may be found in the execution directory. The graphical analysis tool, *paraprof*, may then be invoked to visualize the stored data—the main view window is shown in Fig. 13.1.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 428

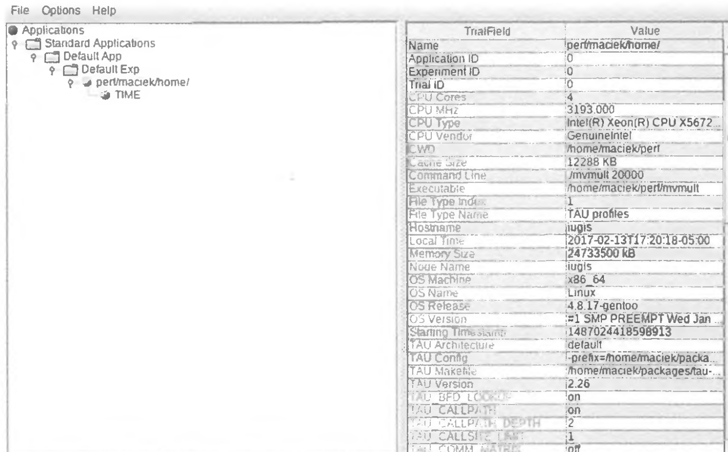


FIGURE 13.1

Main window of *paraprof* analysis GUI.

At the same time *paraprof* opens a second window that visualizes execution phases (see Fig. 13.2).

Moving the mouse cursor over bars representing execution phases provides additional data, while the right-click opens context menus for additional actions. TAU GUI supports many more data views, including histograms, derived metrics, and three-dimensional profiling graphs. Additionally, the data may be presented in text format using the *pprof* utility. The reader is strongly encouraged to explore these options to gain more familiarity with the tool.

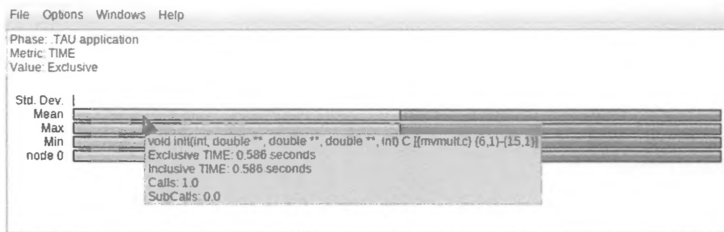


FIGURE 13.2

Paraprof execution-phase window.

13.6 PROFILING IN DISTRIBUTED ENVIRONMENTS

The *gperftools* and *perf* profilers discussed previously were originally developed for use with sequential codes, although there are somewhat more complex ways of using them with parallel programs. TAU, depending on configuration, may be capable of monitoring sequential, OpenMP, and MPI applications. However, there are several software tools explicitly designed for performance monitoring and profiling in distributed environments, including Scalasca [14], VampirTrace [15], and MPE2 [16]. As representative of the capabilities of these tools, this section explores VampirTrace profiling for distributed environments.

VampirTrace is an open-source performance monitoring infrastructure targeting high performance computing (HPC) applications. It provides a means for easily adding timing measurement function calls and performance counters to an application as part of instrumentation. Instrumentation in VampirTrace may be automatic or manual, and can be driven by the choice of programming model (MPI, OpenMP, CUDA, OpenCL, or hybrid), by a third-party package like TAU or Dyninst, or by using the VampirTrace API to insert measurement function calls manually to regions of interest in an HPC application. The output from VampirTrace instrumentation is in an open-source format, called the Open Trace Format, which is readable and analyzable through multiple tools including the proprietary Vampir graphical toolkit. VampirTrace itself is included as part of recent releases of OpenMPI and is frequently found already available on many supercomputers.

For most HPC applications developers, the quickest way to use VampirTrace for performance monitoring is to compile an application using the VampirTrace compiler wrappers: vtcc for C, vtcxx for C++, and vtfort for Fortran. The pingpong.c code illustrated in Code 13.4 is used as an example for MPI code, and the forkjoin.c code illustrated in Code 13.5 as an example for OpenMP code.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include "mpi.h"
5
6  int main(int argc,char **argv)
7  {
8    int rank,size;
9    MPI_Init(&argc,&argv);
10   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11   MPI_Comm_size(MPI_COMM_WORLD,&size);
12
13   if ( size != 2 ) {
14     printf(" Only runs on 2 processes \n");
15     MPI_Finalize(); // this example only works on two processes
16     exit(0);
17   }
18
```

```
19    int count;
20    if ( rank == 0 ) {
21      // initialize count on process 0
22      count = 0;
23    }
24    for (int i=0;i<10;i++) {
25      if ( rank == 0 ) {
26        MPI_Send(&count,1,MPI_INT,1,0,MPI_COMM_WORLD); // send "count" to rank 1
27        MPI_Recv(&count,1,MPI_INT,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE); // receive it back
28        sleep(1);
29        count++;
30        printf(" Count %d\n",count);
31      } else {
32        MPI_Recv(&count,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
33        MPI_Send(&count,1,MPI_INT,0,0,MPI_COMM_WORLD);
34      }
35  }
36
37  if ( rank == 0 ) printf("\t\t\t Round trip count = %d\n",count);
38
39  MPI_Finalize();
40  }
```

Code 13.4. MPI pingpong.c code for illustrating MPI instrumentation using VampirTrace.

```
1   #include <omp.h>
2   #include <unistd.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <math.h>
6
7   int main (int argc, char *argv[])
8   {
9     const int size = 20;
10    int nthreads, threadid, i;
11    double array1[size], array2[size], array3[size];
12
13    // Initialize
14    for (i=0; i < size; i++) {
15      array1[i] = 1.0*i;
16      array2[i] = 2.0*i;
17    }
18
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 431

```
19    int chunk = 3;
20
21 #pragma omp parallel private(threadid)
22   {
23   threadid = omp_get_thread_num();
24   if (threadid == 0) {
25     nthreads = omp_get_num_threads();
26     printf("Number of threads = %d\n", nthreads);
27   }
28   printf(" My threadid %d\n",threadid);
29
30   #pragma omp for schedule(static,chunk)
31   for (i=0; i<size; i++) {
32     array3[i] = sin(array1[i] + array2[i]);
33     printf(" Thread id: %d working on index %d\n",threadid,i);
34     sleep(1);
35   }
36
37   } // join
38
39   return 0;
40 }
```

Code 13.5. Example forkjoin.c code for illustrating instrumentation in OpenMP using VampirTrace.

When compiling C-language-based MPI code with the VampirTrace compiler wrappers, the MPI wrappers can be specified to the VampirTrace wrapper using the `-vt:cc` flag:

```
vtcc -vt:cc mpicc pingpong.c
```

Alternatively, the MPI libraries can be linked in without using the MPI compiler wrapper:

```
vtcc pingpong.c -lmpi
```

Note that in the latter approach the user may have to specify to the compiler where to find the MPI header file (mpi.h) and the MPI library.

While in principal using the VampirTrace compiler wrapper is enough to trigger automatic instrumentation for either MPI, OpenMP, or hybrid MPI+OpenMP applications, it is sometimes necessary to specify the programming model explicitly to the compiler using the `-vt:mpi`, `-vt:mt`, or

-vt:hyb specifications for MPI, OpenMP, and MPI+OpenMP applications respectively. For example, the pingpong.c MPI code (Code 13.4) could be compiled with MPI instrumentation as follows:

```
vtcc -vt:cc mpicc -vt:mpi pingpong.c
```

Likewise, the forkjoin.c OpenMP code in Code 13.5 could be compiled as follows:

```
vtcc -vt:cc gcc -vt:mt -fopenmp forkjoin.c
```

In this OpenMP example, the choice of the underlying compiler was explicitly set to be the GNU compiler (gcc) and the OpenMP library was enabled using the -fopenmp flag.

Once the codes are compiled using the VampirTrace compiler wrappers, they are executed just as they would normally be. However, upon completion of execution, an Open Trace Format file with the name of the code executable will appear in the execution directory. This file contains the measurement information provided by the instrumentation. There are several tools which can read the Open Trace Format file: Figs. 13.3 and 13.4 show Open Trace Files using the Vampir visualizer for the MPI pingpong.c code and the OpenMP forkjoin.c code, respectively.

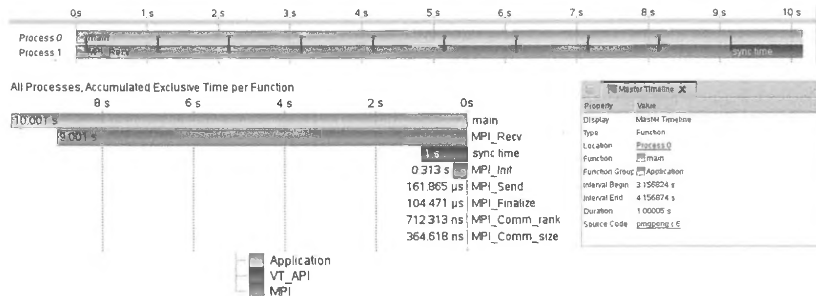


FIGURE 13.3

Instrumentation results from the pingpong.c MPI code (Code 13.4). A phase diagram shows the amount of time spent in application-level code (green (light gray in print versions)), MPI code (red (gray in print versions)), and time associated with the VampirTrace API. The information is shown both individually for each process as a function of time and cumulatively for the entire execution. In the top phase plot where information is shown individually for each process, the messages passed between the processes are illustrated using black lines. Portions of this phase plot can be highlighted and explored in detail with more information on the selected computational phase appearing in the context view labeled "Master Timeline".

FIGURE 13.4

Instrumentation results from the forkjoin.c OpenMP code (Code 13.5) run using eight OpenMP threads. A color-coded computational phase diagram reveals most of the application time spent in the OpenMP loop, except for thread 7 which was idle throughout the computation. The cumulative time spent in each color-coded phase is also reported. Individual phase segments in each thread can be highlighted with more information appearing in the context view labeled "Master Timeline".

Apart from automatic instrumentation based on the programming model, VampirTrace can provide instrumentation via TAU, Dyninst, or manually inserting VampirTrace API calls to the code. These options are specified to the VampirTrace compiler wrapper as follows:

```
vtcc -vt:inst tauinst (For automatic TAU instrumentation)
vtcc -vt:inst dyninst (For automatic Dyninst instrumentation)
vtcc -vt:inst manual (For exclusive manual instrumentation)
```

Manual instrumentation in VampirTrace requires placing two API calls in regions of interest in a source code, VT_USER_START("<user-chosen name>") and VT_USER_END("<user-chosen name>"), and compiling with the flag `-DVTRACE`. To illustrate this, the pingpong.c code (Code 13.4) is modified to add these calls in Code 13.6.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 434

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include "mpi.h"
 5 #include "vt_user.h"
 6
 7 int main(int argc,char **argv)
 8 {
 9   int rank,size;
10   MPI_Init(&argc,&argv);
11   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
12   MPI_Comm_size(MPI_COMM_WORLD,&size);
13
14   if ( size != 2 ) {
15     printf(" Only runs on 2 processes \n");
16     MPI_Finalize(); // this example only works on two processes
17     exit(0);
18   }
19
20   int count;
21   if ( rank == 0 ) {
22     // initialize count on process 0
23     count = 0;
24   }
25   for (int i=0;i<10;i++) {
26     if ( rank == 0 ) {
27       MPI_Send(&count,1,MPI_INT,1,0,MPI_COMM_WORLD); // send "count" to rank 1
28       MPI_Recv(&count,1,MPI_INT,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE); // receive it back
29       VT_USER_START("sleep section");
30       sleep(1);
31       VT_USER_END("sleep section");
32       count++;
33       printf(" Count %d\n",count);
34     } else {
35       MPI_Recv(&count,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
36       MPI_Send(&count,1,MPI_INT,0,0,MPI_COMM_WORLD);
37     }
38   }
39
40   if ( rank == 0 ) printf("\t\t\t Round trip count = %d\n",count);
41
42   MPI_Finalize();
43 }
```

Code 13.6. The pingpong.c code (Code 13.4) has been modified here for manual instrumentation. The VampirTrace API header ("vt_user.h") has been added in line 5 and the calls to VT_USER_START and VT_USER_END have been added surrounding the sleep function call in line 30. The section has been labeled "sleep section".

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 435

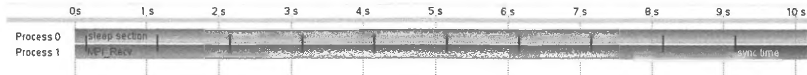


FIGURE 13.5

The computational phase plot for Code 13.6 with manual instrumentation along with compiler instrumentation of the MPI calls. The phase plot is now annotated not only with MPI calls but also with the manually instrumented "sleep section" which appears in the computational phases of process 0.

Code 13.6 can be compiled just as before, but with the `-DVTRACE` flag so that the manually added VampirTrace API calls will be recorded. In this example it is beneficial to combine the manual instrumentation with the MPI instrumentation automatically provided by the compiler, so the `-vt:inst` *manual* specification is not included in the compile command (it would otherwise override all compiler instrumentation):

```
vtcc -vt:cc mpicc -DVTRACE pingpong.c
```

The resulting computational phase plot for each process of Code 13.6 shown in Fig. 13.5 is now annotated with the manually instrumented computational phases labeled "sleep section" as well as the compiler-instrumented MPI phases.

13.7 SUMMARY AND OUTCOMES OF CHAPTER 13

- Performance monitoring is closely associated with application development and optimization. It detects the most frequently executed sections of code and measures the application's resource footprint.
- The act of measuring disturbs the measured system. Performance monitors employ minimally intrusive solutions to collect the performance metrics, leveraging dedicated low-overhead implementations such as hardware event counters whenever available.
- Monitored programs need to be instrumented, i.e., modified through insertion of suitable measurement and result collection functions. This may be accomplished at source level using compiler techniques, at library level by instrumentation, or at executable level. Each of these mechanisms differs in the degree of user involvement, measurement scope and precision, supported features, and intrusiveness.
- One of the fundamental metrics is time. Its measurement may be invoked from the command line using the *time* system utility or by instrumenting an application with timestamp collection functions such as *clock_gettime*.
- Profiling is one of the elementary techniques of performance monitoring. It is used to identify a program's execution hotspots and potentially capture other runtime metrics, such as memory size, communication parameters, and I/O activity. They may be used to classify the program as compute bound, memory bound, or I/O bound.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 436

- Hotspots are parts of code the program spend most of the time executing. Bottlenecks are hotspots that have unduly adverse effects on the application's performance. Program optimization may relocate the bottleneck to another part of the code.
- One of the commonly used general-purpose profilers is provided by the *gperftools* suite. It can detect hotspots and memory management issues without modifications to the source code.
- The *perf_events* and *PAPI* packages are commonly used interfaces accessing hardware event counters. The first may be used from the command line, while the second enables instrumentation of arbitrary application sections.
- *TAU* is an example of an integrated profiling environment that supports multiple instrumentation modes, collection of execution profiles with multiple parameters, custom user probes, performance database management, and both text- and GUI-driven data analysis. It also interoperates with other tools using shared data formats.
- VampirTrace is one of the broadly used distributed profilers that is particularly useful for MPI and OpenMP (or hybrid) program tracing to capture program execution phases and communication activity. The generated traces may be displayed in a proprietary Vampir visualizer or exported to open-source tools such as TAU.

13.8 QUESTIONS AND PROBLEMS

1. Discuss differences between hotspots and bottlenecks. Provide examples to illustrate your answer.
2. Why do hardware event counters often provide a better insight into runtime behavior of an application? What are their limitations?
3. Write a program that estimates the overhead of time measurement using POSIX clocks. Make sure you collect numbers for both "hot" (i.e., initialized) and "cold" (uninitialized) cache scenarios.
4. Consider the following program:

```
1 #include <stdio.h>
2
3 int main() {
4   long sum = 0;
5   for (int i = 0; i < 1000000; i++)
6     if (i&1 != 0) sum += 3*i;
7   printf("sum=%ld\n", sum);
8   return 0;
9 }
```

Instrument the `for`-loop using PAPI to count all conditional branches and taken conditional branches. Estimate the counts and verify your numbers by executing the instrumented code. How do the values change when the program is compiled with optimizations enabled compared to the unoptimized version? Why?
Note: to explain the discrepancies, it may be helpful to look at the generated assembly code. For gcc it can be done using the command:

```
gcc -S -fverbose-asm program.c
```

The resultant assembly listing annotated with variable names will be placed in the file `program.s`.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 437

5. Profiling a program with the *perf* tool produces the following output:

```
Performance counter stats for './a.out':
      14207.022284     task-clock:u (msec)        #   1.000 CPUs utilized
                 0     context-switches:u         #   0.000 K/sec
                 0     cpu-migrations:u           #   0.000 K/sec
            10,301     page-faults:u              #   0.725 K/sec
    50,036,833,663     cycles:u                   #   3.522 GHz
    49,799,684,446     stalled-cycles-frontend:u  #  99.53% frontend
cycles idle
    46,725,530,082     stalled-cycles-backend:u   #  93.38% backend cycles
idle
     1,059,912,928     instructions:u             #   0.02  insn per cycle
                                                  #  46.98  stalled cycles
per insn
       115,260,873     branches:u                 #   8.113 M/sec
            55,407     branch-misses:u            #   0.05% of all
branches
      14.208427535 seconds time elapsed
```

What may be inferred about the code based on the above statistics?

6. Why might correlating different types of metrics supported by tools such as TAU be useful to program optimization? Provide examples.

7. An MPI program that makes a frequent use of MPI_Allreduce calls achieves poor parallel execution performance. Its developer suspects that this is due to load imbalance between the cores. How would you confirm her/his theory using VampirTrace?

REFERENCES

[1] Intel VTune Amplifier 2017, Intel Inc, 2017 [Online]. Available: https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[2] CodeXL Web Page, Advanced Micro Devices, Inc., 2016 [Online]. Available: http://gpuopen.com/compute-product/codexl/.

[3] CodeXL GitHub Repository, 2017 [Online]. Available: https://github.com/GPUOpen-Tools/CodeXL.

[4] AMD Catalyst Software Web Page, Advanced Micro Devices Inc, 2017 [Online]. Available: http://www.amd.com/en-gb/innovations/software-technologies/catalyst.

[5] Nvidia CUDA Toolkit Web Page, Nvidia Inc, 2017 [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[6] D.A. Mills, Computer Network Time Synchronization: The Network Time Protocol, CRC Press, 2006, p. 304 p.

[7] Gperftools Wiki Page, Github, February 2017 [Online]. Available: https://github.com/gperftools/gperftools/wiki.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 438

[8] perf: Linux profiling with performance counters, September 28, 2015 [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page.

[9] PERF_EVENT_OPEN Manual Page, January 10, 2015 [Online]. Available: http://web.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html.

[10] Performance Application Programming Interface, University of Tennessee, February 2017 [Online]. Available: http://icl.cs.utk.edu/papi/.

[11] Intel Math Kernel Library (Intel MKL), Intel Inc, 2017 [Online]. Available: https://software.intel.com/en-us/intel-mkl.

[12] TAU Reference Guide, University of Oregon, November 11, 2016 [Online]. Available: https://www.cs.uoregon.edu/research/tau/tau-referenceguide.pdf.

[13] Paradyn/Dyninst Web Page, University of Meryland and University of Wisconsin Madison, [Online]. Available: http://www.dyninst.org/.

[14] Scalasca Web Page, Juelich Forschungszentrum, Technische Universitaet Darmstadt, German Research School for Simulation Sciences, [Online]. Available: http://www.scalasca.org/.

[15] VampirTrace Web Page, Centre for Information Services and High Performance Computing (ZIH), Dresden University, 2016 [Online]. Available: https://tu-dresden.de/zih/forschung/projekte/vampirtrace.

[16] Performance Visualization for Parallel Programs web page, Laboratory for Advanced Numerical Software at ANL, [Online]. Available: http://www.mcs.anl.gov/research/projects/perfvis/software/index.htm.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 439

CHAPTER

DEBUGGING

14

CHAPTER OUTLINE

14.1 Introduction **421**
14.2 Tools **423**
14.2.1 The GNU Debugger 423
14.2.1.1 Break Points 424
14.2.1.2 Watch Points and Catch Points 425
14.2.1.3 Back Trace 427
14.2.1.4 Setting a Variable 428
14.2.1.5 Threads 430
14.2.1.6 GDB Cheat Sheet 430
14.2.2 Valgrind 430
14.2.3 Commercial Parallel Debuggers 431
14.3 Debugging OpenMP Example: Accessing an Unprotected Shared Variable **433**
14.4 Debugging MPI Example: Deadlock **434**
14.5 Compiler Flags for Debugging **439**
14.6 System Monitors to Aid Debugging **441**
14.7 Summary and Outcomes of Chapter 14 .... **445**
14.8 Exercises **446**
References **450**

14.1 INTRODUCTION

Frequently high performance computing (HPC) practitioners encounter anomalies in application execution that arise from a wide variety of origins, including hardware failures, programming errors, software technical errors, or even the unlikely case of a cosmic ray flipping a bit and interfering with the computation. Tracking the origin of such application execution anomalies is difficult even when using just a simple desktop or laptop computer. On an HPC resource, resolving such an anomaly in an application is compounded many times by the complex interplay between the multiple network, memory, and library components of the supercomputer and the different execution modalities employed. This chapter introduces several techniques and tools for debugging an HPC application and explores several of the more common types of bugs the practitioner will encounter, including deadlocks, races, memory leaks, segmentation faults, and invalid references, among others.

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00014-9
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 440

Grace Brewster Murray Hopper seated at the input console for the UNIVAC I. *Photo by Smithsonian Institution via Wikimedia Commons*

Grace Brewster Murray Hopper was a mathematics professor who became a US Navy rear admiral and strongly promoted and influenced the development of higher-level programming languages at a time when most programming was done in nonportable, machine-specific languages. In addition to her programming language and compiler work, which served as the genesis of the common business-oriented language (COBOL), she was a senior developer on the first commercial electronic computer, the UNIVAC 1. In her own words, "the most important thing I've accomplished, other than building the compiler, is training young people". Among numerous other accolades, Grace Hopper received the highest civilian award of the United States, the Presidential Medal of Freedom, posthumously in 2016.

Historically debugging is popularly associated with Grace Hopper, who discovered a moth interfering with a computer's operation while working on the Harvard Mark II electromechanical computer in 1947. The moth was placed in the group's logbook with the caption "First actual case of bug being found", as seen in Fig. 14.1. In a similar story that slightly predates Grace Hopper's experience, mathematician Norbert Wiener was called to diagnose the anomalous behavior of the automatic fire control of a warship gun during World War II. After hearing a description of the specific short circuits that occurred at certain gun muzzle positions, he correctly predicted that a dead mouse would be found in the device and the specific location where it would be found [1].

Not entirely unlike these famous cases of literal debugging, debugging an application on a high performance computer frequently requires a fairly detailed view of the supercomputer software and hardware stack to diagnose the anomaly properly. There are a wide variety of tools that can assist in diagnosing a problem. This chapter begins by introducing the use of the GNU debugger (GDB) and the Valgrind instrumentation framework, and mentioning some of the more prevalent commercial debugger tools. The chapter then uses the tools to explore a series of common bugs found in message-passing interface (MPI) and OpenMP codes. It finishes by enumerating a list of common compiler flags and messages that are helpful in debugging applications, and some available system monitor approaches to debugging.

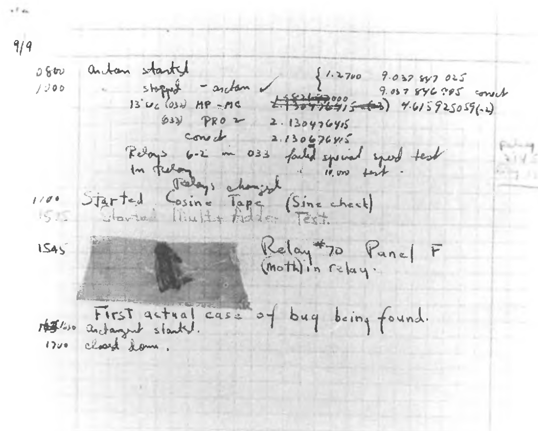


FIGURE 14.1

An example of literal debugging from the Harvard Mark II as recorded by Grace Hopper—a moth found in the machine has been taped to the logbook.

Photo courtesy: Naval Surface Warfare Center, Dahlgren, VA, 1988 via Wikimedia Commons

14.2 TOOLS

The complexity of debugging a code has motivated the development of multiple open-source and proprietary tools to assist the programmer in stepping through a code in execution and enabling the placement of breakpoints where the execution is paused and the memory can be viewed. The most common open-source debugging tools are serial in nature; however, they can be adapted for debugging in parallel, as shown later in this chapter. There are also several commercial debuggers specifically targeting parallel execution on a high performance computer; these are often made available to supercomputing users by system administrators, although the license cost per node may limit the scale at which a commercial debugger can be used. This section introduces two open-source and freely available debugging tools, the GDB and the Valgrind instrumentation framework, and gives some information on a subset of the more common commercial parallel debuggers available.

14.2.1 THE GNU DEBUGGER

The GDB is one of the commonest open-source debuggers available. One commercial debugger (Allinea DDT [2]) even uses the GDB as its engine. The GDB is a command-line debugger invoked on Linux and Unix systems using the command:

gdb <executable name> where the angled brackets are substituted for the executable intended for debugging. This section explores a small but important subset of GDB functionality that is used for the

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 442

```
0001 #include <stdlib.h>
0002 #include <stdio.h>
0003
0004 int main(int argc,char **argv) {
0005   int i;
0006   // Make the local vector size constant
0007   int local_vector_size = 100;
0008
0009   // initialize the vectors
0010   double *a, *b;
0011   a = (double *) malloc(
0012        local_vector_size*sizeof(double));
0013   b = (double *) malloc(
0014        local_vector_size*sizeof(double));
0015   for (i=0;i<local_vector_size;i++) {
0016     a[i] = 3.14;
0017     b[i] = 6.67;
0018   }
0019   // compute dot product
0020   double sum = 0.0;
0021   for (i=0;i<local_vector_size;i++) {
0022     sum += a[i]*b[i];
0023   }
0024   printf("The dot product is %g\n",sum);
0025
0026   free(a);
0027   free(b);
0028   return 0;
0029 }
```

FIGURE 14.2

The example code dotprod_serial.c for exploring the GNU debugger.

debugging examples later in this chapter. To help illustrate GDB commands and usage, the example code of Fig. 14.2 is used. When running GDB on an executable, it is important to let the compiler know that the executable will be used for debugging. This is done by using the "-g" flag when compiling.

14.2.1.1 Break Points

One of the most useful commands in GDB is for setting a break point. A break point is an interruption in the execution of a code, enabling the user to examine the program's state at that moment. There are several ways to set a break point with the GDB, including specifying a function name, line number, file name and line number, a conditional, or even a memory address. Using the code from Fig. 14.2, several of these options are explored in Table 14.1, assuming that the code has been compiled with debug information enabled using the −g flag and the GDB has already started on the executable as indicated.

Information on each of the break points can be queried from the gdb command line using the command *info breakpoints*, as seen in Fig. 14.3. When calling *info breakpoints*, seven quantities are reported: the identifier of the break point, the type of break point, the disposition of the break point, whether or not the break point is enabled, the memory address of where the break point is in the

Table 14.1 Examples of Different Ways to Set a Break Point Using the Code in Fig. 14.2 as an Example

Break Point Command Type	gdb Breakpoint Command	Description
Break by function	break printf	Pauses the execution at line 24
Break by line number	break 17	Pauses the execution at line 17
Break by line number and filename	break dotprod_serial.c:17	Pauses the execution at line 17
Break by conditional	break dotprod_serial.c:16 if i==4	Pauses the execution at line 16 when i equals 4

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400450 <printf@plt>
2       breakpoint     keep y   0x00000000004005ef in main at dotprod_serial.c:17
3       breakpoint     keep y   0x00000000004005ef in main at dotprod_serial.c:17
4       breakpoint     keep y   0x00000000004005cf in main at dotprod_serial.c:16
        stop only if i==4
(gdb)
```

FIGURE 14.3

Information on all the break points in Table 14.1 set for the code in Fig. 14.2.

program, and where the break point is in terms of the file name and line number. While only break points have been discussed up to this point, two similar types of pausing points, called watch points and catch points, are discussed in the following subsection.

The disposition of a break point indicates whether it will be deleted when reached, or kept. This is often useful when setting a break point inside a for-loop so that the same break point is not repeatedly hit. A break point can be disabled by using the *disable* command followed by the break point identifier. For example, entering *disable 2* in the command line would disable break point number 2. It can be reenabled by using the command *enable* 2. Break points can be deleted altogether by using the *delete* command followed by the break point identifier. The disposition of a break point can also be changed by using the *enable* command. For example, if break point number 3 should be disabled after being hit once, the command *enable once 3* is used. To set a break point with the disposition to be deleted when hit, the *tbreak* command is used following the syntax of Table 14.1. These four useful break point commands, *enable, disable, delete,* and *tbreak*, are illustrated in Fig. 14.4.

While the setting of the break point by itself is frequently useful in helping to deduce control flow, it is usually most useful in examining the variables at that moment in the program's state. This can be done using the *print* command, illustrated in Fig. 14.5. Note that for the execution to begin after setting the break point in Fig. 14.5, the command *run* must be issued. Once the break point is reached, the execution will pause and the variables can then be examined via *print.*

14.2.1.2 Watch Points and Catch Points

Watch points and catch points are similar in nature to break points, but are conditional upon some variable being written to or some prespecified event like catching a C++ exception. To set a watch

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 444

```
(gdb) disable 2
(gdb) enable once 3
(gdb) delete 1
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep n   0x00000000004005ef in main at dotprod_serial.c:17
3       breakpoint     dis  y   0x00000000004005ef in main at dotprod_serial.c:17
4       breakpoint     keep y   0x00000000004005cf in main at dotprod_serial.c:16
        stop only if i==4
(gdb) tbreak dotprod_serial.c:24
Temporary breakpoint 5 at 0x40067b: file dotprod_serial.c, line 24.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep n   0x00000000004005ef in main at dotprod_serial.c:17
3       breakpoint     dis  y   0x00000000004005ef in main at dotprod_serial.c:17
4       breakpoint     keep y   0x00000000004005cf in main at dotprod_serial.c:16
        stop only if i==4
5       breakpoint     del  y   0x000000000040067b in main at dotprod_serial.c:24
(gdb)
```

FIGURE 14.4

Beginning with the break points in Fig. 14.3, the commands disable, enable, delete, and tbreak are used to alter the enablement of a break point, disposition of a break point, deletion a break point, and setting of a temporary break point, respectively.

```
(gdb) tbreak 17
Temporary breakpoint 1 at 0x4005ef: file dotprod_serial.c, line 17.
(gdb) run
Starting program: /home/andersmw/learn/a.out

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdfc8) at dotprod_serial.c:17
17          b[i] = 6.67;
(gdb) print i
$1 = 0
(gdb) print a[i]
$2 = 3.1400000000000001
(gdb) print b[i]
$3 = 0
(gdb)
```

FIGURE 14.5

Example of using a temporary break point at line 17 of Fig. 14.2 and then examining the values of the variables inside the break point. Notice that the a[0] element has been initialized while the b[0] element has not yet been initialized, indicating that the break point pauses before the specified break point line is executed.

point, the command *watch* followed by the expression to watch is entered into the gdb command line. For example, to watch for changes to the value of the sum variable in Fig. 14.2, the command *watch sum* would be issued to the gdb command line once the variable sum was in the current context at line 20. This is illustrated in Fig. 14.6. Information on watch points can be obtained issuing the *info watchpoints* command illustrated in Fig. 14.7.

```
(gdb) b 20
Breakpoint 1 at 0x40061b: file dotprod_serial.c, line 20.
(gdb) r
Starting program: /home/andersmw/learn/a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:20
20        double sum = 0.0;
(gdb) watch sum
Hardware watchpoint 2: sum
(gdb) continue
Continuing.
Hardware watchpoint 2: sum

Old value = 6.9533558074263132e-310
New value = 0
main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:21
21        for (i=0;i<local_vector_size;i++) {
(gdb) continue
Continuing.
Hardware watchpoint 2: sum

Old value = 0
New value = 20.9438
main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:21
21        for (i=0;i<local_vector_size;i++) {
(gdb) 
```

FIGURE 14.6

A demonstration of setting a watch point on the variable sum from Fig. 14.2. A break point is set at line 20 so the variable is in the current memory context. Then the watch point is issued using issuing the command "watch sum". Each time the sum variable is written to, the execution will pause. The "continue" command is used to resume execution. The watch point is hit twice in this example. The abbreviations for "break", "b", and "run", "r", are also used. Command abbreviations are included in the GDB cheat sheet.

```
(gdb) info watchpoints
Num     Type           Disp Enb Address            What
2       hw watchpoint  keep y                      sum
        breakpoint already hit 2 times
(gdb) 
```

FIGURE 14.7

Information on watch points can be obtained issuing the "info watchpoints" command.

14.2.1.3 Back Trace

When the execution has paused in the debugger, an overview of the callers leading to the present point in the execution can be revealed using the *back trace* command. As the example in Fig. 14.2 only has one call (main), any back trace using that example would only give one frame, or call stack member. To better illustrate the back trace command, the example of Fig. 14.2 is modified to include another function as seen in Fig. 14.8.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 446

```
0001 #include <stdlib.h>
0002 #include <stdio.h>
0003
0004 void initialize(double *a, double *b,int local_vector_size)
0005 {
0006   int i;
0007   for (i=0;i<local_vector_size;i++) {
0008     a[i] = 3.14;
0009     b[i] = 6.67;
0010   }
0011 }
0012
0013 int main(int argc,char **argv) {
0014   int i;
0015   // Make the local vector size constant
0016   int local_vector_size = 100;
0017
0018   // initialize the vectors
0019   double *a, *b;
0020   a = (double *) malloc(
0021        local_vector_size*sizeof(double));
0022   b = (double *) malloc(
0023        local_vector_size*sizeof(double));
0024
0025   initialize(a,b,local_vector_size);
0026
0027   // compute dot product
0028   double sum = 0.0;
0029   for (i=0;i<local_vector_size;i++) {
0030     sum += a[i]*b[i];
0031   }
0032   printf("The dot product is %g\n",sum);
0033
0034   free(a);
0035   free(b);
0036   return 0;
0037 }
```

FIGURE 14.8

Example code for exploring the back trace command.

By setting a break point at line 8 in the initialize function of Fig. 14.8, the call stack for that point in the execution can be revealed using the back trace command, as shown in Fig. 14.9.

The call stack can be traversed using the *up* and *down* commands, enabling the user to exit or enter function calls and examine the variables and memory in those calls. The up and down commands, illustrated in Fig. 14.10, can be followed by a number to traverse several call stack frames with a single command.

14.2.1.4 Setting a Variable

Using GDB it is possible to set a variable during execution and continue execution using that variable. This capability is achieved using the *set* command, illustrated in Fig. 14.11. After setting a break point

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 447

```
(gdb) break 8
Breakpoint 1 at 0x40059e: file dotprod_serial.c, line 8.
(gdb) run
Starting program: /home/andersmw/learn/a.out

Breakpoint 1, initialize (a=0x601010, b=0x601340, local_vector_size=100)
    at dotprod_serial.c:8
8           a[i] = 3.14;
(gdb) backtrace
#0  0x000000000040059e in initialize (a=0x601010, b=0x601340, local_vector_size=100)
    at dotprod_serial.c:8
#1  0x0000000000400643 in main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:25
(gdb)
```

FIGURE 14.9

Illustration of the back trace command for showing the call stack. A break point is set in the code from Fig. 14.8 at line 8 and the code is executed to that point. Issuing the back trace command reveals a call stack with two frames: the execution frame in the initialize function (frame #0), and the calling frame (frame #1) back to the main routine.

```
(gdb) backtrace
#0  0x000000000040059e in initialize (a=0x601010, b=0x601340, local_vector_size=100)
    at dotprod_serial.c:8
#1  0x0000000000400643 in main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:25
(gdb) up
#1  0x0000000000400643 in main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:25
25        initialize(a,b,local_vector_size);
(gdb) list
20        a = (double *) malloc(
21             local_vector_size*sizeof(double));
22        b = (double *) malloc(
23             local_vector_size*sizeof(double));
24
25        initialize(a,b,local_vector_size);
26
27        // compute dot product
28        double sum = 0.0;
29        for (i=0;i<local_vector_size;i++) {
(gdb) down
#0  initialize (a=0x601010, b=0x601340, local_vector_size=100) at dotprod_serial.c:8
8           a[i] = 3.14;
(gdb) list
3
4       void initialize(double *a, double *b,int local_vector_size)
5       {
6         int i;
7         for (i=0;i<local_vector_size;i++) {
8           a[i] = 3.14;
9           b[i] = 6.67;
10        }
11      }
12
(gdb)
```

FIGURE 14.10

Example of traversing the call stack frames using the up and down commands. Beginning with the back trace from Fig. 14.9, the up command is issued moving the debugger context outside the initialize function to the main routine at line 25 of Fig. 14.8. The "list" command is useful in printing a few lines of the source code from the current context to screen. The down command is then issued and the debugger context is returned to the initialize function.

```
(gdb) break 17
Breakpoint 1 at 0x4005ef: file dotprod_serial.c, line 17.
(gdb) run
Starting program: /home/andersmw/learn/a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffdfb8) at dotprod_serial.c:17
17          b[i] = 6.67;
(gdb) set var i=99
(gdb) continue
Continuing.
The dot product is 0
[Inferior 1 (process 12264) exited normally]
(gdb) 
```

FIGURE 14.11

After setting a break point inside the initialization for-loop in the code from Fig. 14.2, the value of the variable i is set to 99, forcing the loop to exit when execution is resumed. Using the "set var" command, the execution can be steered inside the debugger.

inside the initialization *for*-loop in the code from Fig. 14.2, the value of the variable *i* is set to 99, forcing the loop to exit when execution is resumed. Using the "set var" command, the execution can be steered inside the debugger.

14.2.1.5 Threads

For multithreaded applications such as OpenMP, the GDB enables switching context between threads as well as applying debugger commands to all threads. The *info threads* command will list the threads along with a thread identifier. The debugger can switch between threads by issuing the *thread* command followed by the thread identifier.

To explore the thread debugging functionality in GDB, the OpenMP dot product example in Fig. 14.12 is used. The environment variable OMP_NUM_THREADS is set to four and the GDB is started in the normal way: *gdb <executable name>*. Stepping through the code and examining the private variables of each thread is illustrated in Fig. 14.13. A break point is placed at line 23 of the code in Fig. 14.12. The debugger notifies the user of the creation of three additional threads upon running, making a total of four threads as expected. Once at the break point, the command "*info threads*" lists the threads available. When issuing "*info threads*", the asterisk that appears next to the thread number indicates which thread context is active in the debugger. The private variable *i* is printed for each thread and the debugger context is switched between the threads using the "thread" command.

14.2.1.6 GDB Cheat Sheet

A brief summary of some of the more important GDB commands is listed in Table 14.2, along with their functions and abbreviations.

14.2.2 VALGRIND

The Valgrind tool suite [3] provides several very important tools for debugging applications, especially in the context of memory errors and thread data races. The suite consists of the tools shown in Table 14.3.

```
0001 #include <stdio.h>
0002 #include <omp.h>
0003
0004 int main ()
0005 {
0006   const int n = 30;
0007   int   i,chunk;
0008   double a[n], b[n], result = 0.0;
0009
0010   /* Some initializations */
0011   chunk = 5;
0012   for (i=0; i < n; i++) {
0013       a[i] = i * 3.14;
0014       b[i] = i * 6.67;
0015   }
0016
0017 #pragma omp parallel for        \
0018      default(shared) private(i) \
0019      schedule(static,chunk)     \
0020      reduction(+:result)
0021
0022     for (i=0; i < n; i++)
0023         result += (a[i] * b[i]);
0024
0025   printf("Final result= %f\n",result);
0026 }
```

FIGURE 14.12

OpenMP dot product code to illustrate the GDB capability with threads.

Like the GDB, it is best practice to compile the executable with debugging information using the −g flag to provide the most information. Valgrind usage is simple: the executable is passed to Valgrind after passing the desired suite tool or check to perform. For example, the command

valgrind −tool=helgrind <program executable> would run the Helgrind tool for finding data race conditions on a specified program executable, such as an OpenMP code. If no tool is specified, Valgrind will run the Memcheck tool. Memcheck is one of the most widely used tools for identifying memory errors.

14.2.3 COMMERCIAL PARALLEL DEBUGGERS

There are a number of commercial parallel debuggers providing debugging support for C, C++, and Fortran-based codes for a wide variety of programming models and hardware architectures, including general-purpose graphics processing units and many integrated core architectures. A list of some of the more widely used parallel commercial debuggers available is provided in Table 14.4.

Each of the debuggers in Table 14.1 has a graphical user interface (GUI) for examining the state of each process or thread in a parallel execution. Several provide detection for memory leak or other memory errors. It is also common now to provide a replay capability whereby the execution state of the entire program is recorded for later playback. This can be especially useful in debugging the

```
(gdb) break 23
Breakpoint 1 at 0x40093c: file dotproduct.c, line 23.
(gdb) run
Starting program: /home/andersmw/learn/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff73d1700 (LWP 44176)]
[New Thread 0x7ffff6bd0700 (LWP 44177)]
[New Thread 0x7ffff63cf700 (LWP 44178)]

Breakpoint 1, main._omp_fn.0 () at dotproduct.c:23
23              result += (a[i] * b[i]);
(gdb) info threads
  Id   Target Id         Frame
  4    Thread 0x7ffff63cf700 (LWP 44178) "a.out" main._omp_fn.0 () at dotproduct.c:23
  3    Thread 0x7ffff6bd0700 (LWP 44177) "a.out" main._omp_fn.0 () at dotproduct.c:23
  2    Thread 0x7ffff73d1700 (LWP 44176) "a.out" main._omp_fn.0 () at dotproduct.c:23
* 1    Thread 0x7ffff7fdc7c0 (LWP 44172) "a.out" main._omp_fn.0 () at dotproduct.c:23
(gdb) print i
$1 = 0
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff73d1700 (LWP 44176))]
#0  main._omp_fn.0 () at dotproduct.c:23
23              result += (a[i] * b[i]);
(gdb) print i
$2 = 5
(gdb) thread 3
[Switching to thread 3 (Thread 0x7ffff6bd0700 (LWP 44177))]
#0  main._omp_fn.0 () at dotproduct.c:23
23              result += (a[i] * b[i]);
(gdb) print i
$3 = 10
(gdb) thread 4
[Switching to thread 4 (Thread 0x7ffff63cf700 (LWP 44178))]
#0  main._omp_fn.0 () at dotproduct.c:23
23              result += (a[i] * b[i]);
(gdb) print i
$4 = 15
(gdb) 
```

FIGURE 14.13

GNU debugger using threads. The code in Fig. 14.12 is executed in the GDB, where the environment OMP_NUM_THREADS is set to be four.

so-called Heisenbugs, which disappear when attempting to trap them. The startup options for TotalView include both a replay capability and memory debugging, as seen in Fig. 14.14. The entire program state can be viewed and toggled between each process or thread, as illustrated for TotalView in Fig. 14.15.

Commercial parallel debuggers provide excellent debugging support, but often at a significant license cost that becomes prohibitive as the number of nodes increases. For this reason, supercomputing centers frequently have an upper limit on the number of nodes across which such commercial debuggers will function. Application users debugging on scales above this limit will often have to revert to some of the other tools discussed in this chapter.

Table 14.2 Brief Summary of Some Major GDB Commands

Command	Abbreviation	Function
Run	r	Begins execution in the debugger
continue	c	Continues execution in the debugger after a pause
quit	q	Quits the debugger
break	b	Sets a break point
watch		Sets a watch point
backtrace	bt	Prints the call stack
set variable	set var	Sets a variable value
thread	t	Switches to a different thread identifier
list	l	Lists source code near the present stopping point

Table 14.3 Tools in the Valgrind Tool Suite

Tool	Description
Memcheck	Reports memory errors, including memory leaks and access to memory that is not yet allocated
Cachegrind	Identifies the number of cache misses
Callgrind	Extends cachegrind with some additional information
Massif	Heap profiler
Helgrind	Debugger for finding data race conditions
DRD	Multithread debugging for C and C++ programs

Table 14.4 Some of the More Widely Used Parallel Commercial Debuggers

Commerical Debugger	Notable Capabilities
TotalView [4]	Support for OpenMP, MPI, OpenACC, CUDA
Allinea DDT [2]	Support for OpenMP, Pthreads, MPI, CUDA
Intel Parallel Debugger [5]	Support for multicore debugging

14.3 DEBUGGING OPENMP EXAMPLE: ACCESSING AN UNPROTECTED SHARED VARIABLE

One of the most common errors made by OpenMP programmers is accessing an unprotected shared variable; an example is shown in Fig. 14.16. A correct version of this example is given in Fig. 14.17.

If the code in Fig. 14.16 is run using Valgrind, the data race on the variable *sum* is immediately identified:

valgrind −tool=helgrind ./a.out

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 452

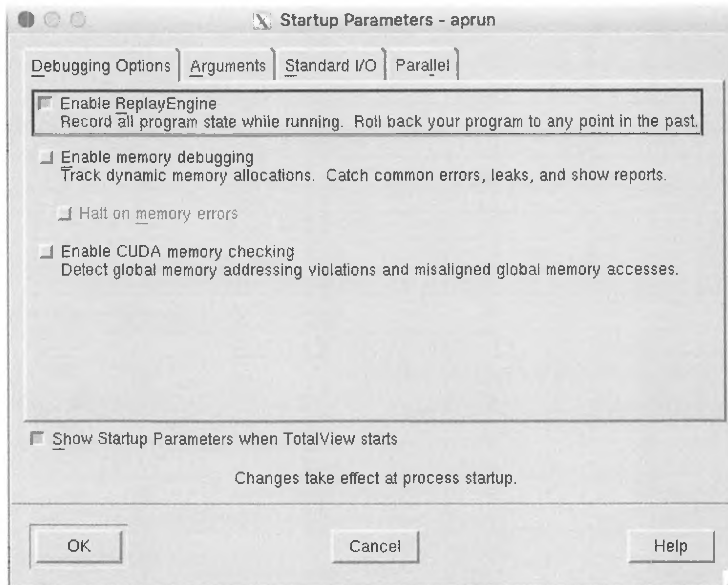


FIGURE 14.14

Startup options for a two-process MPI SendRecv example using TotalView. Important options include enabling replay and memory debugging.

Valgrind produces a warning of a data race in the code *bug.c* (Fig. 14.16); this warning is shown in Fig. 14.18, and it even correctly indicates the line number where the problem occurs. This experiment could also have been conducted using the GDB to observe the race condition as different threads attempt to write to the variable *sum* concurrently.

14.4 DEBUGGING MPI EXAMPLE: DEADLOCK

A common error in MPI programming is a deadlock, where competing requests completely impede their fulfillment and the program cannot proceed. An example is given in Fig. 14.19, and the situation is rectified in Fig. 14.20. Deadlocks like this can be difficult to debug, as they result in the program execution hanging without error message or additional output.

This deadlock can be easily identified using a debugger. Although the GDB is a serial debugger, one simple and straightforward way to debug this parallel application is to launch the GDB for each process. There are two ways to do this.

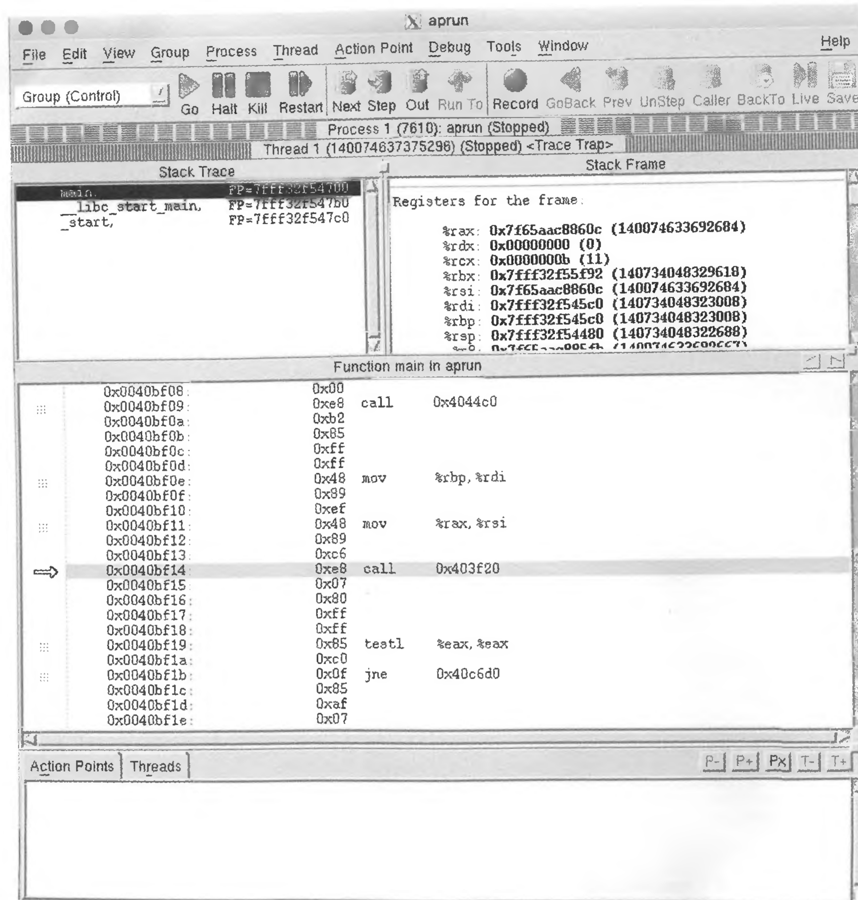


FIGURE 14.15

The TotalView GUI for examining and stepping through the program state on each process or thread.

The first approach involves launching an xterm window for each process and using the debugger in each to investigate the problem. This is illustrated in Fig. 14.21. While this will work on some clusters, many are not configured to allow xterm windows to be launched from the compute nodes.

The second approach does not require launching an xterm window and will work on nearly all clusters. However, it requires adding a few lines of code to what is being debugged. This additional

```
#include <stdio.h>
#include <omp.h>

int main () {
  int i;
  int sum =  ;
#pragma omp parallel for
  for (i= ;i<   ;i++) {
    sum += i;
  }
  printf("          ",sum);
}
```

FIGURE 14.16

Example of unprotected access of a shared variable: bug.c. The shared variable is "sum", and running this with more than one OpenMP thread will result in both incorrect and inconsistent results.

```
#include <stdio.h>
#include <omp.h>

int main () {
  int i;
  int sum =  ;
#pragma omp parallel for reduction(+:sum)
  for (i= ;i<   ;i++) {
    sum += i;
  }
  printf("          ",sum);
}
```

FIGURE 14.17

Corrected version of the code in Fig. 14.16.

```
==30110== Possible data race during write of size 4 at 0x5845800 by thread #1
==30110== Locks held: none
==30110==    at 0x4E4E638: gomp_barrier_wait_end (bar.c:40)
==30110==    by 0x4E4C681: gomp_team_start (team.c:805)
==30110==    by 0x4E48999: GOMP_parallel (parallel.c:167)
==30110==    by 0x400725: main (bug.c:7)
```

URE 14.18

tput from Valgrind when debugging the code bug.c in Fig. 14.16.

e prints the process identifier (PID) for each process to allow the GDB to be attached to that cess. A "while" loop is added to pause the execution of the code until a debugger can be attached to ı process. The deadlock code modified for debugging with the GDB is presented in Fig. 14.22; the that has been added is seen in lines 17–24.

'o debug in parallel with the GDB, the code in Fig. 14.22 is run as normal on two processes, i.e., un −np 2<executable name>. The PIDs then print and the execution will pause, as illustrated in 14.23.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 455

```
0001 #include <stdio.h>
0002 #include <stdlib.h>
0003 #include <mpi.h>
0004
0005 int main(int argc, char* argv[]) {
0006   const int n = 10000;
0007   int *x, *y, nprocs,proc_id,i;
0008   int tag1 = 10;
0009   int tag2 = 20;
0010   MPI_Status status;
0011   MPI_Request send_request, recv_request;
0012
0013   MPI_Init(&argc, &argv);
0014   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
0015   MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);
0016
0017   x = (int *) malloc(sizeof(int)*n);
0018   y = (int *) malloc(sizeof(int)*n);
0019
0020   // this example only works on two processes
0021   if (nprocs != 2) {
0022     if (proc_id == 0) {
0023       printf("This only works on 2 processes\n");
0024     }
0025     MPI_Finalize();
0026     return 0;
0027   }
0028
0029   if (proc_id == 0) {
0030     // only process 0 does this part
0031     for (i=0;i<n;i++) x[i] = 314159;
0032
0033     MPI_Send(x, n, MPI_INT, 1, tag2, MPI_COMM_WORLD);
0034     MPI_Recv(x, n, MPI_INT, 1, tag1, MPI_COMM_WORLD,&status);
0035
0036     printf(" Process %d received value %d\n",proc_id,x[0]);
0037   } else {
0038
0039     MPI_Send(y, n, MPI_INT, 0, tag1, MPI_COMM_WORLD);
0040     MPI_Recv(y, n, MPI_INT, 0, tag2, MPI_COMM_WORLD,&status);
0041   }
0042
0043   free(x);
0044   free(y);
0045
0046   MPI_Finalize();
0047   return 0;
0048 }
```

FIGURE 14.19

Example of a deadlock: two competing MPI_Send requests block the communication progress and the execution hangs.

Once the PIDs are known, the GDB can be attached to each process. This is done by logging on to the node(s) where the processes are waiting and launching the GDB for each PID waiting on that node, as illustrated in Fig. 14.24.

[andersmw@cutter: ~$ gdb attach 17331

[andersmw@cutter: ~$ gdb attach 17332

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 456

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
  const int n =       ;
  int *x, *y, nprocs,proc_id,i;
  int tag1 =   ;
  int tag2 =   ;
  MPI_Status status;
  MPI_Request send_request, recv_request;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

  x = (int *) malloc(sizeof(int)*n);
  y = (int *) malloc(sizeof(int)*n);

  // this example only works on two processes
  if (nprocs !=  ) {
    if (proc_id ==  ) {
      printf("[illegible]");
    }
    MPI_Finalize();
    return  ;
  }

  if (proc_id ==  ) {
    // only process 0 does this part
    for (i= ;i<n;i++) x[i] =      ;

    MPI_Send(x, n, MPI_INT,  , tag2, MPI_COMM_WORLD);
    MPI_Recv(x, n, MPI_INT,  , tag1, MPI_COMM_WORLD,&status);

    printf("[illegible]",proc_id,x[ ]);
  } else {

    MPI_Recv(y, n, MPI_INT,  , tag2, MPI_COMM_WORLD,&status);
    MPI_Send(y, n, MPI_INT,  , tag1, MPI_COMM_WORLD);
  }

  free(x);
  free(y);

  MPI_Finalize();
  return  ;
}
```

FIGURE 14.20

A corrected version of the deadlock in Fig. 14.19.

Note that there is no need to attach the debugger to the process in the same directory as in the original executable. This will start a GDB for each process. Each process will still be in the *while* loop in line 23 of Fig. 14.22, so it will be necessary to change the value of the "*i*" variable to proceed with the debugging. Running *back trace* in one of the debuggers shows that the "*i*" variable is not in the current call stack frame, but two frames above the current execution frame. This is shown in Fig. 14.24.

```
andersmw@cutter:~/learn$ mpirun -np 2 xterm -e gdb ./a.out
```

```
gdb
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...(no debugging symbols found)...done.
(gdb)
```

```
gdb
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...(no debugging symbols found)...done.
(gdb)
```

FIGURE 14.21

Example of launching two serial debuggers via xterm to debug the deadlock in Fig. 14.19. While this will work on some clusters, many will not be configured to allow this type of operation.

To set the "*i*" variable to some value other than 0 and thereby break out of the *while* loop in line 23, the debugger frame is changed to frame #2 and the "*i*" is set to 1 using the *set variable* command in GDB. This is done in both debugger command lines, and is illustrated in Fig. 14.25.

The code is now running in parallel within two different instances of the GDB. Generally it is best practice to set any desired break points prior to issuing the *continue* command in Fig. 14.25. However, in this deadlock example the debugger will be used to establish why the code hangs by simply stopping the execution of both debuggers using *control-c* and then issuing the *back trace* command in each debugger, as illustrated in Fig. 14.26.

The back traces from both debugger instances gives the call stacks for the two hanging processes, indicating that they are both waiting on account of blocking send calls resulting in a deadlock.

The debugger allows the MPI application developer to query the behavior of a parallel application directly, place break points and watch points, and traverse the call stack and memory to diagnose problems quickly at large scales. While in this example a GDB was attached to each process, this is probably not feasible when debugging with thousands of processes. In such a case the debugger can be attached to only a relevant subset of processes by appropriately modifying the code inserted to print out PIDs and wait, as shown in lines 17–24 of Fig. 14.22.

14.5 COMPILER FLAGS FOR DEBUGGING

Compiler warnings are a significant resource to assist in debugging an application. Specific command-line options for the compiler can be used to check for common mistakes programmers make. In Table 14.5 a summary of command-line options for the GNU, Intel, LLVM, and PGI compilers is presented, along with the associated action they invoke in the context of debugging.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 458

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
  const int n =       ;
  int *x, *y, nprocs,proc_id,i;
  int tag1 =   ;
  int tag2 =   ;
  MPI_Status status;
  MPI_Request send_request, recv_request;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);

  // Wait for attach
  i =  ;
  char hostname[    ];
  gethostname(hostname, sizeof(hostname));
  printf("[illegible]", getpid(), hostname);
  fflush(stdout);
  while (  == i)
      sleep( );

  x = (int *) malloc(sizeof(int)*n);
  y = (int *) malloc(sizeof(int)*n);

  // this example only works on two processes
  if (nprocs !=  ) {
    if (proc_id ==  ) {
      printf("[illegible]");
    }
    MPI_Finalize();
    return  ;
  }

  if (proc_id ==  ) {
    // only process 0 does this part
    for (i= ;i<n;i++) x[i] =       ;

    MPI_Send(x, n, MPI_INT,  , tag2, MPI_COMM_WORLD);
    MPI_Recv(x, n, MPI_INT,  , tag1, MPI_COMM_WORLD,&status);

    printf("[illegible]",proc_id,x[ ]);
  } else {

    MPI_Send(y, n, MPI_INT,  , tag1, MPI_COMM_WORLD);
    MPI_Recv(y, n, MPI_INT,  , tag2, MPI_COMM_WORLD,&status);
  }

  free(x);
  free(y);

  MPI_Finalize();
  return  ;
}
```

FIGURE 14.22

Deadlock example modified for attaching to the debugger. Lines 17–24 have been added to print the PIDs and then wait for the debugger to be attached to those processes.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 459

```
andersmw@cutter:~/learn$ mpirun -np 2 ./a.out
PID 17331 on cutter ready for attach
PID 17332 on cutter ready for attach
```

FIGURE 14.23

Running the code in Fig. 14.22 results in the PIDs for the processes printing to screen. The execution then pauses due to the "while" loop in line 23 of Fig. 14.22.

```
(gdb) backtrace
#0  0x00007fa6cebcbdfd in nanosleep () at ../sysdeps/unix/syscall-template.S:81
#1  0x00007fa6cebcbc94 in __sleep (seconds=0)
    at ../sysdeps/unix/sysv/linux/sleep.c:137
#2  0x0000000000400c55 in main (argc=1, argv=0x7ffe6f24c2d8) at deadlock.c:24
```

FIGURE 14.24

The call stack upon attaching the GDB to one of the PIDs. Note that the call frame where the "*i*" variable and *while* loop are found is frame #2, or two frames up from the execution frame (frame #0).

```
(gdb) up 2
#2  0x0000000000400c55 in main (argc=1, argv=0x7ffe6f24c2d8) at deadlock.c:24
24              sleep(5);
(gdb) list
19          char hostname[256];
20          gethostname(hostname, sizeof(hostname));
21          printf("PID %d on %s ready for attach\n", getpid(), hostname);
22          fflush(stdout);
23          while (0 == i)
24              sleep(5);
25
26          x = (int *) malloc(sizeof(int)*n);
27          y = (int *) malloc(sizeof(int)*n);
28
(gdb) set var i=1
(gdb) continue
```

FIGURE 14.25

The variable "*i*" is set to 1 to break out of the *while* loop, pausing execution in the code of Fig. 14.22. This is done by changing the execution frame to be where the variable "*i*" is in the current context ("up 2"), resetting "*i*" to be 1 ("set var i=1"), and resuming execution.

14.6 SYSTEM MONITORS TO AID DEBUGGING

Many clusters employ monitoring software to inspect the status of node hardware and obtain information about the currently executing workload. The former may be as simple as verification that the node is responsive to remote commands, but may also include measurement of temperatures of critical components (they typically rise under increased load) or even access to low-level built-in sensors that monitor other physical aspects of the hardware (supply voltages, fan speeds, etc.). The latter is primarily concerned with the utilization of available central processing units (CPUs) (see Fig. 14.27), but

MPI Process 0

```
^C
Program received signal SIGINT, Interrupt.
0x00007fa6cebd62a7 in sched_yield () at ../sysdeps/unix/syscall-template.S:81
81      ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) backtrace
#0  0x00007fa6cebd62a7 in sched_yield () at ../sysdeps/unix/syscall-template.S:81
#1  0x00007fa6c93da772 in psmi_mq_wait_internal (do_lock=0, status=0x0, ireq=0x7ff
e6f24bde8) at psm_mq.c:279
#2  0x00007fa6c93da772 in psmi_mq_wait_internal (ireq=0x7ffe6f24bde8)
    at psm_mq.c:314
#3  0x00007fa6c93bd61f in amsh_mq_send (len=40000, ubuf=0x153c6a0, tag=20, flags=<
optimized out>, epaddr=0x1517498, req=0x7fa6cf551ef0, mq=0x14c3468)
    at am_reqrep_shmem.c:2799
#4  0x00007fa6c93bd61f in amsh_mq_send (mq=0x14c3468, epaddr=0x1517498, flags=<opt
imized out>, tag=20, ubuf=0x153c6a0, len=40000) at am_reqrep_shmem.c:2847
#5  0x00007fa6c93daa4b in __psm_mq_send (mq=<optimized out>, dest=<optimized out>,
 flags=<optimized out>, stag=<optimized out>, buf=<optimized out>, len=<optimized
out>) at psm_mq.c:393
```

MPI Process 1

```
^C
Program received signal SIGINT, Interrupt.
0x00007f1c104ef690 in __psmi_poll_internal (ep=0x1ebf538,
    poll_amsh=poll_amsh@entry=1) at psm.c:499
499     }
(gdb) backtrace
#0  0x00007f1c104ef690 in __psmi_poll_internal (ep=0x1ebf538, poll_amsh=poll_ams
h@entry=1) at psm.c:499
#1  0x00007f1c104ed7a6 in psmi_mq_wait_internal (do_lock=0, status=0x0, ireq=0x7
ffcffbfeac8) at psm_mq.c:279
#2  0x00007f1c104ed7a6 in psmi_mq_wait_internal (ireq=0x7ffcffbfeac8)
    at psm_mq.c:314
#3  0x00007f1c104d061f in amsh_mq_send (len=40000, ubuf=0x1f0a280, tag=429496731
5, flags=<optimized out>, epaddr=0x1e6b2e8, req=0x7f1c16664ef0, mq=0x1e88458)
    at am_reqrep_shmem.c:2799
#4  0x00007f1c104d061f in amsh_mq_send (mq=0x1e88458, epaddr=0x1e6b2e8, flags=<o
ptimized out>, tag=4294967315, ubuf=0x1f0a280, len=40000)
    at am_reqrep_shmem.c:2847
#5  0x00007f1c104eda4b in __psm_mq_send (mq=<optimized out>, dest=<optimized out
>, flags=<optimized out>, stag=<optimized out>, buf=<optimized out>, len=<optimi
zed out>) at psm_mq.c:393
```

FIGURE 14.26

The back trace from both debugger instances after pausing execution via control-c to find out why the program is hanging. The call stacks for both processes indicate that they are both waiting (frames #1 and #2) on account of blocking send calls resulting in a deadlock.

may also provide other important statistics such as fraction of workload spent executing in user and system modes, amount of used and free memory, volume of data transferred in input/output operations, network traffic level, available disk space, and others. The monitoring relies on lightweight daemons executing in the background on every node that sample and collect the required information at regular intervals (e.g., every minute). This information is aggregated on a dedicated server and available to users through a commonly accessible interface, such as a webpage. Commonly used system monitors include Nagios [6] and Ganglia [7].

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 461

Table 14.5 Command-Line Options Used for Debugging for GNU, Intel, LLVM, and PGI Compilers

Action	Gcc	icc	clang	pgcc
Enable pointer bounds checking (R)	-fcheck-pointer-bounds	-check-pointers-mpx=rw		-Mbounds
Enable address sanitizer (R)	-fsanitize=address		-fsanitize=address	
Enable thread sanitizer (R)	-fsanitize=thread		-fsanitize=thread	
Enable leak sanitizer (R)	-fsanitize=leak		-fsanitize=leak	
Enable undefined behavior sanitizer (R)	-fsanitize=undefined		-fsanitize=undefined	
Enable all common warning types (S)	-Wall	-Wall	-Wall	-Minform=warn
Warn if the code does not strictly comply with ANSI C or ISO C++ (S)	-pedantic		-pedantic	-Xa
Warn on use of uninitialized variables (S)	-Wuninitialized	-Wuninitialized	-Wuninitialized	
Warn when local variable shadows another variable (S)	-Wshadow	-Wshadow	-Wshadow	
Warn if comparison between signed and unsigned integer may produce wrong result (S)	-Wsign-compare	-Wsign-compare	-Wsign-compare	
Warn if undefined identifier is used in preprocessor directive (S)	-Wundef		-Wundef	
Warn when undeclared function is used or declaration does not specify a type (S)	-Wimplicit	-Wmissing-declarations -Wmissing-prototypes	-Wimplicit	

ANSI, American National Standards Institute.
Note: Actions annotated with (R) denote that error conditions are indicated during runtime, while (S) produces a warning during static analysis of the code (compilation). Options in shaded cells do not accurately reflect the semantics of the Gcc option in the same row.

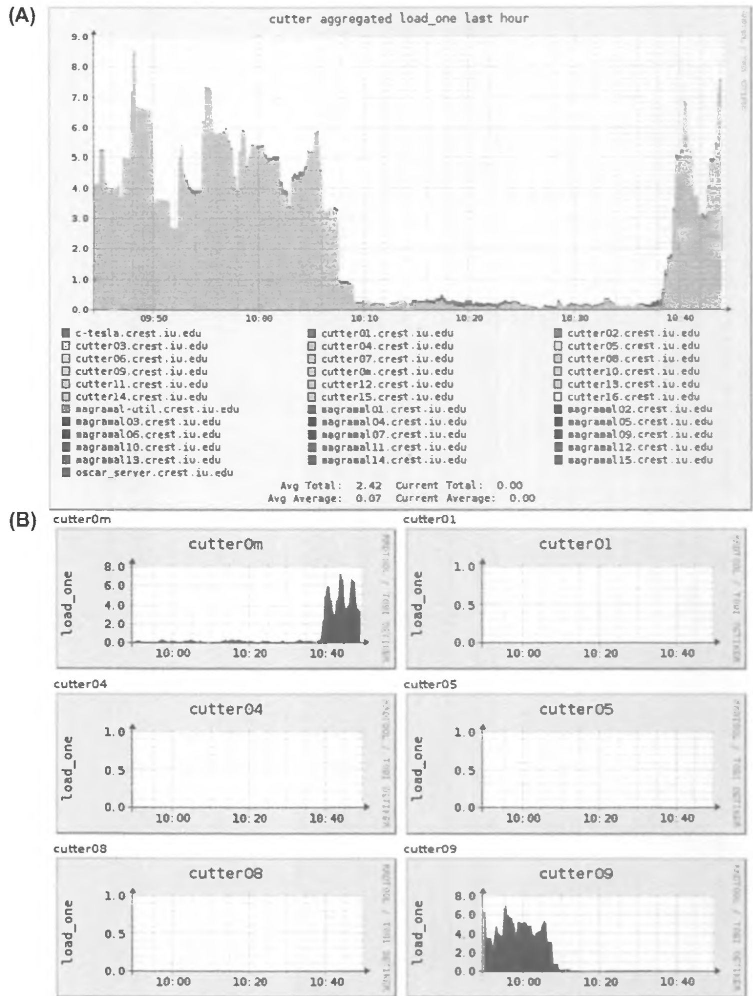


FIGURE 14.27

Example snapshot of processor load produced by Ganglia and presented as (A) composite graph for all monitored nodes and (B) individual nodes (only a fragment shown due to space constraints).

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 463

Since sampling is performed at a relatively coarse resolution to minimize the impact of monitoring on the primary workload execution, only limited analysis is possible. However, coupling the execution of a debugged application with a graphical representation of system status may frequently provide clues that would be otherwise difficult to obtain. Any load imbalances during application execution are immediately visible. If the load is expected to be uniform by algorithm design but in reality is asymmetric, this immediately identifies locations (nodes) that require closer inspection. This may arise from logical flaws in the code, but may also be caused by an incorrectly terminated job that previously executed on the same node or a system service that got out of control. Threads stuck in a spin lock usually exhibit CPU load close to 100%, while idling threads (such as those waiting for tasks to execute) have a minimal CPU utilization. Large load changes observed in multithreaded programs may suggest incorrectly designed critical sections or improper locking mechanisms. Monitoring memory usage may explain random performance fluctuations caused, for example, by approaching the point of exhaustion of physical memory. While a debugger will certainly catch a failed memory allocation call, it often will not be able to establish whether the failure occurred after prolonged execution with a large memory footprint or was a result of a spurious allocation request. Observation of network traffic may help identify undesirable hotspots for algorithms with an expectation of uniform communication patterns. While many of the system-monitor-inspired approaches are related to performance debugging, harnessing them for conventional debugging may help focus on the true cause of faults faster. They also provide much-needed sanity checks to verify that the startup environment for application execution matches the programmer's expectations.

14.7 SUMMARY AND OUTCOMES OF CHAPTER 14

- Tracking the origin of a parallel application execution anomaly on a supercomputer is generally much more difficult than debugging a serial application.
- Debugging an application on a high performance computer frequently requires a fairly detailed view of the supercomputer software and hardware stack to diagnose the anomaly properly.
- Several open-source and commercial debugging tools and suites have been developed to assist the debugging process.
- There are several commercial parallel debuggers which support MPI and OpenMP codes.
- There are several open-source serial debuggers and tool suites which can be used to debug MPI and OpenMP codes. In the case of MPI, they may require attaching several serial debuggers to a simulation.
- The GDB provides multiple tools for debugging a code and enabling the user to step through the code and call stack, as well as viewing variables and changing their values.
- The GDB also provides support for debugging codes with multiple threads.
- The Valgrind suite of tools provides six major tools for debugging applications, including rectifying data races and memory leaks.
- Multiple serial debuggers can be attached to an MPI execution to conduct parallel debugging.
- There is significant compiler support for debugging through specific flags to enable pointer bounds checking and other memory checking.
- System monitors provide an independent way to examine program execution and match that to the programmer's expectations.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 464

14.8 EXERCISES

1. The following code allocates and initializes a two-dimensional array for a send buffer in connection with an MPI code. However, it has memory problems: an invalid write and a memory leak. Use Valgrind to identify and fix the invalid write and memory leak.

```
 1 #include <stdio.h>
 2 #include <stdlib.>
 3
 4 int main(int argc,char **argv) {
 5
 6   int comm_count = 20;
 7   int numfields = 10;
 8   int length = 159;
 9
10   double **send_buffer = (double **) malloc(comm_count*sizeof(double *));
11   for (int p=0;p<comm_count;p++) {
12     send_buffer[p] = (double *) malloc(numfields*length*sizeof(double));
13   }
14
15   // Copy data into the send buffer
16   for (int p=0;p<comm_count;p++) {
17     for (int fields=0;fields<numfields;fields++) {
18       for (int i=0;i<=length;i++) {
19         send_buffer[p][i + length*numfields] = 3.14159;
20       }
21     }
22   }
23
24   return 0;
25 }
```

2. The following code uses OpenMP to compute $a_3 = sin(a_1 + a_2)$ where a_1, a_2, a_3 are arrays of length 20.

```
 1 #include <omp.h>
 2 #include <unistd.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5 #include <math.h>
 6
 7 int main (int argc, char *argv[])
 8 {
 9  const int size = 20;
10  int nthreads, threadid, i;
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 465

```
11   double array1[size], array2[size], array3[size];
12
13   // Initialize
14   for (i=0; i < size; i++) {
15     array1[i] = 1.0*i;
16     array2[i] = 2.0*i;
17   }
18
19   int chunk = 3;
20
21 #pragma omp parallel private(threadid)
22   {
23   threadid = omp_get_thread_num();
24   if (threadid == 0) {
25     nthreads = omp_get_num_threads();
26     printf("Number of threads = %d\n", nthreads);
27   }
28   printf(" My threadid %d\n",threadid);
29
30   #pragma omp for schedule(static,chunk)
31   for (i=0; i<size; i++) {
32     array3[i] = sin(array1[i] + array2[i]);
33     printf(" Thread id: %d working on index %d\n",threadid,i);
34     sleep(1);
35   }
36
37   } // join
38
39   return 0;
40 }
```

Run the code using four OpenMP threads. Use the GDB to perform the following operations. Put a hardware watch point on the variable *nthreads*. Which thread ID stops at this hardware watch point? Does the debugger thread ID correspond to the *threadid* variable in line 23 of the code?

3. The following code creates a new communicator, and within that communicator sends its rank to its new communicator neighbor. However, it has a bug. This code works properly when run on between one and four processes, but hangs when using anything more than four processes. Use the tools and techniques from the chapter to debug why this code fails on five or more processes. Then fix the problem.

```
1 #include <mpi.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 466

```
 5 int main(int argc,char *argv[])
 6 {
 7   int myid,numprocs;
 8
 9   MPI_Init(&argc,&argv);
10   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
11   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
12
13   int color = myid%2;
14   MPI_Comm new_comm;
15   MPI_Comm_split(MPI_COMM_WORLD,color,myid,&new_comm);
16
17   int new_id,new_nodes;
18   MPI_Comm_rank(new_comm,&new_id);
19   MPI_Comm_size(new_comm,&new_nodes);
20
21   printf(" Rank %d Numprocs %d New id %d New nodes %
     d\n",myid,numprocs,new_id,new_nodes);
22
23   int right = (new_id + 1) % new_nodes;
24   int left = new_id - 1;
25   if (left < 0)
26    left = new_nodes - 1;
27
28   int buffer[2],buffer2[2];
29   MPI_Status status;
30   buffer[0] = myid;
31   buffer[1] = rand();
32
33   MPI_Sendrecv(buffer, 2, MPI_INT, right, 123,
34     buffer2, 2, MPI_INT, right, 123, new_comm, &status);
35
36   printf(" Rank %d received %d\n",myid,buffer2[0]);
37
38   MPI_Finalize();
39   return 0;
40 }
```

4. The following code hangs when run on two processes.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include "mpi.h"
4
```

```
 5 int main(int argc, char* argv[]) {
 6   const int n = 100000;
 7   int x[n], y[n], np, id,i;
 8   int tag1 = 19;
 9   int tag2 = 20;
10   MPI_Status status;
11
12   MPI_Init(&argc, &argv);        /* Initialize MPI */
13   MPI_Comm_size(MPI_COMM_WORLD, &np); /* Get number of processes */
14   MPI_Comm_rank(MPI_COMM_WORLD, &id); /* Get own identifier */
15
16   /* Check that we run on exactly two processes */
17   if (np != 2) {
18    if (id == 0) {
19      printf("Only works on 2 processes\n");
20    }
21    MPI_Finalize();     /* Quit if there is only one process */
22    exit(0);
23   }
24
25   if (id == 0) {   /* Process 0 does this */
26     for (i=0;i<n;i++) x[i] = 314159;
27
28     MPI_Send(&x, n, MPI_INT, 1, tag2, MPI_COMM_WORLD);
29     MPI_Recv(&x, n, MPI_INT, 1, tag1, MPI_COMM_WORLD,&status);
30
31     printf(" Process %d received value %d\n",id,x[0]);
32   } else {
33     for (i=0;i<n;i++) y[i] = 137035;
34     MPI_Send(&y, n, MPI_INT, 0, tag1, MPI_COMM_WORLD);
35     MPI_Recv(&y, n, MPI_INT, 0, tag2, MPI_COMM_WORLD,&status);
36   }
37
38   MPI_Finalize();
39   exit(0);
40 }
```

a. Why does it hang?

b. When size of variable n in line 6 is changed to be much smaller, the code does not hang any more. Try this: set $n = 10$ in line 6. What does process 0 print to screen? Why?

c. Why does the code not hang when variable n is small?

d. Fix the problem so that the code works for any n size. What prints to screen as the result now? Why?

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 468

REFERENCES

[1] F. Conway, J. Siegelman, Dark Hero of the Information Age: In Search of Norbert Wiener, The Father of Cybernetics, s.l., Basic Books, 2006.
[2] Allinea, Allinea DDT, [Online]. http://www.allinea.com/products/ddt.
[3] Valgrind Tool Suite, [Online]. valgrind.org.
[4] RogueWave Software, TotalView for HPC, 2016 [Online]. http://www.roguewave.com/products-services/totalview.
[5] Intel, Intel Parallel Debugger Extension, [Online]. https://software.intel.com/en-us/articles/parallel-debugger-extension.
[6] Nagios: The Industry Standard in IT Infrastructure Monitoring, [Online]. https://www.nagios.org.
[7] Ganglia Monitoring System, [Online]. http://ganglia.info/.

CHAPTER

ACCELERATOR ARCHITECTURE 15

CHAPTER OUTLINE

15.1 Introduction 451
15.2 A Historic Perspective 454
15.2.1 Coprocessors 456
15.2.1.1 Intel 8087 457
15.2.1.2 Motorola MC68881 459
15.2.2 Accelerators in Processor I/O Space 461
15.2.3 Accelerators With Industry-Standard Interfaces 462
15.3 Introduction to Graphics Processing Units 464
15.4 Evolution of Graphics Processing Unit Functionality 466
15.5 Modern Graphics Processing Unit Architecture 471
15.5.1 Compute Architecture 471
15.5.2 Memory Implementation 474
15.5.3 Interconnects 475
15.5.4 Programming Environment 476
15.6 Heterogeneous System Architecture 477
15.7 Summary and Outcomes of Chapter 15 480
15.8 Problems and Questions 480
References 481

15.1 INTRODUCTION

The design of the modern processor involves multiple trade-offs focused on optimizing the functionality, performance, energy consumption, and manufacturing cost. The final product is a compromise between the supported feature set, physical constraints, and a projected retail price. Since CPUs must execute a very broad range of workload types, their instruction sets are as generic as possible to enable reasonable performance for most applications. While additional specialized function units could be and sometimes are incorporated on processor dies to enable hardware support for specific computation types, this increases the final chip and case size. The function units may also require additional input/output (I/O) pins for dedicated communication links or memory banks, and increased

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00015-0

Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 470

die size results in greater probability of the occurrence of manufacturing defects. All these factors have nonlinear effects on the final product price, which often renders such enhancements prohibitive.

Practical accelerators explore different functionality, power requirements, and resultant price points to offer complementary features to existing processors, albeit without trying to optimize execution performance for all anticipated application profiles. In high performance computing (HPC) accelerators are typically employed to increase the computational throughput (most often expressed in terms of floating-point operations per second), although at a cost of programmability. Control logic used by accelerators is often incompatible with the existing processor instruction set architecture (ISA), forcing the application developers to invest their time in mastering custom programming languages, language extensions, or wrapper libraries provided by the vendor to enable access to accelerator features. More often than not, naïve usage of accelerators without a working knowledge of their control and data paths and other details of internal architecture does not yield the desired results expected by interpolation from raw peak performance of the underlying hardware modules and general application traits measured on conventional processors. Thus programming of such *heterogeneous systems* that include accelerators working side by side with regular processors still presents many challenges to the uninitiated.

To improve portability, accelerators are frequently attached to the remainder of the system using industry-standard interfaces, such as Peripheral Component Interconnect (PCI) Express [1] (see Fig. 15.1). This permits the incorporation of accelerated hardware in practically any machine that is equipped with such an interface, has sufficient power budget to supply energy to the accelerator, and,

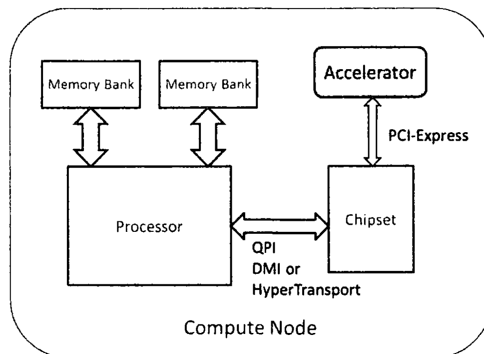


FIGURE 15.1

Typical placement of an accelerator in a conventional compute node. Modern processors often incorporate Peripheral Component Interconnect (PCI) Express endpoints on the die, thereby not relying on the chipset as a necessary component to achieve accelerator connectivity.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 471

CHAPTER

THE ESSENTIAL OPENACC 16

CHAPTER OUTLINE

16.1 Introduction **483**
  16.1.1 CUDA 485
  16.1.2 OpenCL 485
  16.1.3 C++ AMP 486
  16.1.4 OpenACC 486
16.2 OpenACC Programming Concepts **487**
16.3 OpenACC Library Calls **489**
16.4 OpenACC Environment Variables **491**
16.5 OpenACC Directives **492**
  16.5.1 Parallel Construct 493
  16.5.2 Kernels Construct 495
  16.5.3 Data Management 497
  16.5.4 Loop Scheduling 501
  16.5.5 Variable Scope 504
  16.5.6 Atomics 504
16.6 Summary and Outcomes of Chapter 16 .... **506**
16.7 Questions and Problems **506**
References **508**

16.1 INTRODUCTION

As discussed in Chapter 15, graphics processing units (GPUs) are currently one of the most dominant accelerator types employed in high performance computing. In contrast to conventional multicore processors, however, their programming is a much more complex task. The main reason for this stems from the relatively young age of GPU technology, resulting in a dearth of mature programming tools and environments. Various aspects of the technology are constantly being improved and modified, which further complicates the development of general-purpose programming approaches and compilers. Compared to conventional hardware, the accelerators also use a diametrically different execution model. While for many practical purposes each core on a multicore CPU could be considered a separate context of execution, the same is not true for a thread ensemble running on a GPU core. This is particularly apparent in cases of performance loss due to *branch divergence*, when a

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00016-2
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 472

subset of threads follows a different code path than the others as a result of a conditional instruction. Conventional processor architecture in combination with an optimizing compiler makes many implicit components of program execution (register allocation, cache management, data consistency enforcement, optimization of branches, instruction reordering, speculative execution, and many others) transparent to the user, who is free to focus on fleshing out the essential program algorithms and data structures in a high-level programming language. In GPUs many details of the architecture still need to be explicitly addressed by a programmer who is interested in extracting the highest level of performance. Due to the much larger number of execution resources and also the stronger emphasis on parallelism, resource allocation and management become far more critical to achieving a good level of performance. These often have to take into account the physical structure, count, and resource limits on GPUs, especially if many computational kernels with different memory footprints and performance characteristics need to be scheduled concurrently. Since data locality references play a critical role in maximizing the performance and GPU memory capacity is traditionally undersized compared to that of the host machine, efficient scheduling of data offloads adds another dimension to the complexity of managing the computations on an accelerator. Note that offload speeds are usually constrained by the available bandwidth of the PCI Express bus, potentially resulting in significant latencies when transferring large amounts of data. To offer any advantage over a nonaccelerated model of computation, these costs would have to be amortized by performance gains over the entire course of an application execution. Moreover, the question of what is the right placement for a specific kernel in a heterogeneous architecture is not always easy to answer. It has to be weighed against the individual programmer's experience in GPU code development, familiarity with the architectural features of the target GPU, programming tools available, and ported algorithm characteristics. Even then it may turn out that due to unforeseen overheads or latencies the speed-up gained through execution on an accelerator does not present any practical advantage compared to conventional hardware. This directly affects programmers' productivity: their time would have likely been better spent developing and optimizing a multicore implementation of the algorithm, or even better linking with an optimized external library providing the required functionality. Finally, to take advantage of both worlds, one might attempt to balance the computation across all available execution resources in the system. While potentially yielding the best performance, this approach is also the most difficult to manage. Strong disparities between the execution environments involved make the predictable scheduling of computations very difficult to attain, save for the most trivial and well-characterized problems.

Initially, GPU programs leveraged three-dimensional graphics application programming interfaces (APIs) such as OpenGL [1] and DirectX [2] to perform operations on vectors and dense matrices, since these were natively supported by the graphics pipeline. One of the first algorithms accelerated on a GPU was matrix multiplication using 8-bit (with 16-bit internal precision) fixed-point arithmetic published in 2001 [3]. To trick the graphics hardware into performing the desired operations, the authors used two textures corresponding to the input matrices and mapped multiple copies of them on to the interior of a cube, keeping one parallel and the other perpendicular to the projection plane. The partial products obtained through multitexturing in modulate mode were summed on to the front face of the cube using blending in orthographic view (to avoid perspective distortions). The final result (image) was then retrieved using GPU-to-CPU memory copy. The reader will immediately notice that this method of performing computations is not very practical. To provide a more convenient programming environment, a number of custom interfaces specialized for GPUs and in some cases

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 473

targeting general heterogeneous platforms were developed throughout the 2000s. As the feature sets of newer GPUs grew richer and after the introduction of new architectural capabilities (programmable shaders, double-precision floating points, support for dynamic parallelism, etc.), many of these interfaces were revised to include the appropriate support for added extensions. It is not uncommon for many of these APIs to undergo several specification revisions over the relatively short span of their existence, the newest of which frequently require recent versions of graphics hardware to provide the full set of operational features. A brief overview of several popular toolkits with different programming models, supported features, portability, and scope is presented below.

16.1.1 CUDA

This widespread proprietary GPU programming toolkit, originally known as the Compute Unified Device Architecture (CUDA) [4], only works with devices manufactured by Nvidia, including the GeForce, Quadro, and Tesla families. Frequently used high performance computing languages such as C, C++, and Fortran are supported through compiler extensions and a runtime library. For the C family of languages Nvidia provides `nvcc`, a low level virtual machine-based compiler, while Fortran support is available from the Portland Group's (PGI) CUDA Fortran compiler. The programming environment is supplemented by libraries optimized for specific tasks, such as fast Fourier transform computation, basic linear algebra subprograms, random number generation, dense and sparse solvers, graphs analytics, and game physics simulation. CUDA has several performance-oriented features that are typically not available through standard graphics-based interfaces, such as scattered memory reads, unified memory access, fast on-GPU shared memory access, improved speeds of offload and state retrieval, additional data types, mixed-precision computing, supplementary integer and bit-wise operations, and profiling support. As of June 2017, the most recent revision of the toolkit is 8.0.

16.1.2 OPENCL

Open Computing Language [5], initially released in 2009 by the non-profit Khronos consortium, is an open standard attempting to define a unified heterogeneous programing framework. It provides an API on top of the C language (ISO/IEC 9899:1999) and C++14 (starting with revision 2.2) that supports using the target device's memory and processing elements (PEs) for program execution. Execution in a heterogeneous environment places substantial constraints on language features that are permitted—for example, recursion, type identification, go-to statements, virtual functions, exceptions, and function pointers may not be used at all or only with severe limitations. Device vendors determine how and which PEs are actually offered to the user. OpenCL permits up to four levels of memory hierarchy to be implemented by the device: global memory (large, but with substantial latency), read-only memory (small and fast, but writable by the host only), local memory shared by a subset of PEs, and per PE private memory (e.g., registers). Corresponding qualifiers (`global`, `local`, `constant`, `private`) are integrated with the language and understood by the compiler when used in variable declarations. Functions executing on accelerators are marked with the `kernel` attribute and accept argument declarations tagged with the address space qualifiers listed above. Kernels defined as source code may be compiled in runtime by the appropriate online compiler if the platform is *full-profile* compliant; otherwise an offline, platform-specific compilation is used (*embedded profile*). Besides explicitly defined kernels, devices may provide

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 474

built-in functions that are enumerated and offered by OpenCL. The framework supports execution synchronization at three levels: workgroup, subgroup, and command. Revision 2.2—3 of the OpenCL specification was released in May 2017.

16.1.3 C++ AMP

Developed by Microsoft, C++ Accelerated Massive Parallelism [6] is a compiler and set of extensions to C++ that enable the acceleration of C++ applications on platforms that support various forms of data-parallel execution. The accelerator does not necessarily have to be an external device such as a GPU; it could be integrated on the same die as the main CPU, or even be an extension of the main processor's industry-standard architecture, such as streaming single-instruction multiple data (SIMD) extensions or advanced vector extensions provided by some members of the ×86 processor family. Its device model assumes that the accelerator may be equipped with a private memory that is not accessible to the host, or that both host and device share the same memory. The C++ AMP runtime performs or avoids memory copies as required by a particular implementation. The framework defines two types of function restriction specifiers, `cpu` and `amp`, the latter of which marks the relevant code for execution on the accelerator. Functions tagged in this way must conform to the C++ subset that is permitted by the underlying hardware type. Accelerators are represented by `accelerator` objects with an associated logical *view* (more than one view per accelerator is possible) that implement command buffers for computational tasks to be processed by the accelerator. Commands may be submitted for execution immediately or deferred; completion of the accelerator workload may be synchronous (blocking) or asynchronous, using future-based markers for a single task or task group. Data types are based on n-dimensional arrays with related n-dimensional *extent* (determining array bounds) and *index* objects (referring to a specific element). To exercise control over data copying and caching with minimal overhead, *array views* are provided that permit access to a segment of a relevant array. Array views may be accessed locally or in a different coherence domain, implying the necessary data copies for the latter. C++ AMP also supports a range of atomic operations and a `parallel_for_each` construct to launch parallel operations. The current revision of the specification is v1.2, released in 2013.

16.1.4 OPENACC

The Open Accelerator framework [7], also known as "directives for accelerators," differs from the approaches described above in that it attempts to simplify the accelerator programming interface significantly, making code development for GPUs and other attached devices more approachable to a casual developer. It also focuses on better code and performance portability across different platforms. The initial OpenACC specification was created by PGI, CAPS Entreprise, Cray, and Nvidia in 2011. Since then the group has been joined by national labs and multiple industry and academic members, including AMD, Pathscale, and Sandia and Oak Ridge National Laboratories. Since the directive-based approach requires compiler support, commercial tools from PGI (support for multiple target platforms with OpenACC compatibility version 2.5) and Cray (for Cray systems only) are available. Several open-source compilers have also been developed, including OpenUH from University of

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 475

Houston [8], OpenARC provided by Oak Ridge National Laboratory [9], and GCC's experimental OpenACC v2.0a support starting with version 5.1, to be further refined in the GCC 6 release series. Since OpenACC resembles another directive-based parallel programming framework, OpenMP, it is expected that the two environments will eventually be combined and share a single programming specification. The most recent (October 2015) revision of the OpenACC API is 2.5. Its essential features are discussed in more detail in the remainder of this chapter.

16.2 OPENACC PROGRAMMING CONCEPTS

OpenACC supports offloading of designated parts of the program on to accelerator devices connected to the local host computer. Segments of code that may benefit from parallel execution must be explicitly identified by the programmer through relevant directives, or *pragmas* in C and C++, and specially formatted comments in Fortran. Automatic detection of the offloadable sections of program is not supported. The applied method is portable between different CPU types, supported accelerator devices, and underlying operating systems. The details of initialization of accelerator hardware and suitable functions responsible for parallel code execution, management of workload offload, and result retrieval from the accelerator are hidden from the programmer and performed implicitly by the compiler and runtime system. OpenACC currently does not support automatic workload distribution across multiple accelerator devices, even if such are available on the same host machine. Similarly to OpenMP, the directives are simply ignored if the relevant functionality is not supported or not enabled in the compiler.

The execution of the user application is controlled by the host, which nominally follows most of the control flow within the program and initiates transfer of work and data constituting the identified parallel regions to the accelerator. For these code segments, the host may be involved in the allocation of sufficient memory on the device to accommodate the computational kernel's dataset, performing the relevant data transfer between the host and accelerator memory (frequently over the direct memory access or DMA channel), sending the executable code, marshalling and forwarding the input arguments for the parallel region, queuing the code for execution, waiting for completion, and finally fetching the computation results and releasing the memory allocated on the device. Accelerators typically support several levels of parallelism: coarse grain, referring to parallel execution on multiple execution resources, fine grain, involving one of multiple threads within a PE, and function unit level, which exposes SIMD or vector operations within each fine-grain execution unit. In OpenACC these levels are matched respectively by *gang*, *worker*, and *vector* parallelism, as illustrated in Fig. 16.1. The accelerator device executes a number of gangs, each of which contains one or more workers. In turn, a worker may take advantage of available vector parallelism by executing SIMD or vector instructions.

Execution of a compute region on the accelerator starts in so-called *gang-redundant* (GR) mode, in which each gang has a single worker executing the same code. Once the control flow in the program reaches the region marked for parallel execution, the execution switches to *gang-partitioned* (GP) mode, where the work performed by different iterations of one loop or multiple loops is distributed across the gangs, but still with only one worker active in each gang. In both these scenarios program execution proceeds in *worker-single* mode; similarly, if only one lane of vector processing is used by the worker, the program operates in *vector-single* mode. If the parallel region or its section has been

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 476

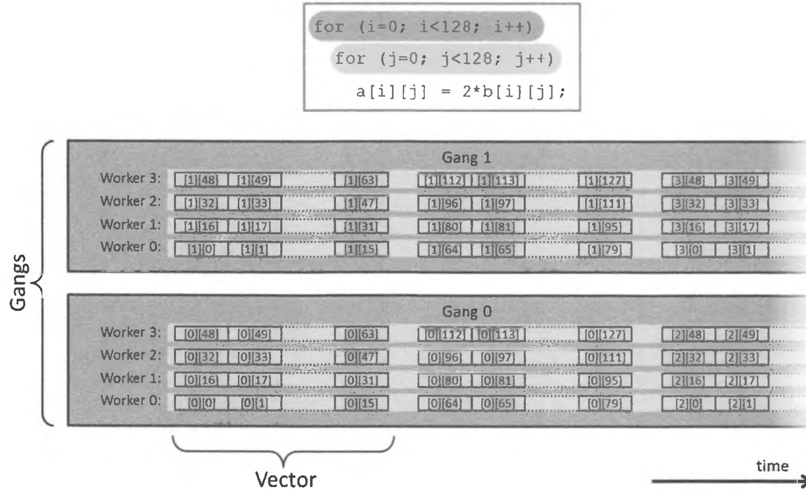


FIGURE 16.1

Example mapping of nested loop iterations on to OpenACC parallelism levels with 2 gangs, 4 workers, and 16 vector lanes. The numeric indices of the accessed matrix element in a specific iteration are shown in square brackets. In this case the outer loop is partitioned across gangs, while the inner loop iterations are divided among workers and vector lanes.

marked for worker-level work sharing, all workers in a gang are activated and the execution continues in *worker-partitioned* mode (WP). Note that parallel regions may enable GP and WP modes at the same time, which causes distribution of available work among all workers in all gangs. A similar distinction applies to vector parallelism: it may be enabled on a per loop or loop nest basis to partition the parallel operations across available SIMD or vector units, thus executing in *vector-partitioned* (VP) mode. VP mode for the specific portion of workload may be activated concurrently with any combination of gang and worker modes.

Explicit synchronization involving barriers or locks across gangs, workers, and vector operations is discouraged. Due to differences between OpenACC implementations and accelerator architectures, some of the gangs may not even begin to execute before others complete. A similar observation applies to workers and vector lanes: since scheduling of worker or vector operations is not always defined deterministically, a specific workload synchronization method that works on one accelerator architecture may lead to a deadlock on another.

Both hosts and accelerators use the concept of a thread, albeit with some differences. Host threads are closely tied to processor execution units, such as cores or hyperthread slots, depending on the

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 477

actual architecture. What constitutes an accelerator core strongly depends on the accelerator type or even the particular implementation of the same device type. For example, AMD demarcates core boundaries on its GPUs differently from Nvidia. OpenACC defines the accelerator thread as a single lane of a single worker in a gang; this unambiguously corresponds to a single parallel execution context. Most accelerator threads can operate asynchronously from host threads. The framework permits submitting the work units to one or more *activity queues* on the device. Operations entered in a single queue will execute in submission order, but operations stored in different activity queues may execute in arbitrary order. The usage of other multithreading environments on the host, such as OpenMP, concurrently with OpenACC is generally unrestricted, although users should take care to avoid oversubscription of execution resources if OpenACC code regions are also scheduled to run on the host processors.

The conscientious OpenACC programmer must be aware of the consequences of the memory model exposed by the framework. Many accelerators, especially PCI Express attached GPUs, are equipped with separate memories from that of the host computer. It means that the host is incapable of directly accessing the device memory and, conversely, the device cannot efficiently access the host memory. Data movement between the two memory pools has to be orchestrated through other means, such as DMA. The programmer must take this into account when writing portable OpenACC code, since the overhead of scheduling and performing a data transfer between host and accelerator memory usually impacts the overall execution performance and may vary from instance to instance. When computing on a large amount of data, the programmer must also be aware of memory size limitations, which are typically much more restrictive on the accelerator side. The datasets accessed by the application must be appropriately partitioned into pieces that may individually fit in the device memory, in some cases imposing changes on the computational algorithm. Data structures containing raw pointers to data in the host memory may also have to be redesigned. Many GPUs utilize a weak memory model in which operations between accelerator threads are performed in arbitrary order unless synchronized by a memory fence, thus potentially producing different results for multiple runs of the same code. Similar considerations apply to unified memory architectures or those offering shared memory space between the host and the accelerator or multiple accelerators. Explicit synchronization to ensure that updates to shared data are fully carried out before they are accessed by the consumer entity is strongly recommended.

16.3 OPENACC LIBRARY CALLS

OpenACC provides a number of predefined values and library functions that may be invoked from user applications. Note that in general none of these functions is required to create fully functional OpenACC programs. They are used in situations when additional information has to be retrieved from the system or explicit management of runtime functions may yield better execution performance. Specifications subdivide the library interfaces into five major sections: definitions, device-oriented functions, asynchronous queue management, device functionality tests, and memory management. Since application of many of these requires an in-depth understanding of host–accelerator interactions, only a small subset of the available interfaces is discussed below.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 478

Since actual OpenACC implementations may conform to different revisions of the specification, one of the macros provided by the OpenACC library may be used to test for the provided functionality. It is called _OPENACC and expands to a six-digit decimal number, in which the first four digits denote the year and the remaining two the month of the specification release date on which the library is based. The _OPENACC macro may be used to enable conditional compilation of code segments that rely on more recently introduced features.

The OpenACC library definitions comprise prototypes of runtime functions and internal data types used by the library that specifically describe runtime function arguments as well as enumerations that identify accelerator types or variants of asynchronous request queue management. The commonly used runtime calls include the following.

```
int acc_get_num_devices(acc_device_t devtype);
```

This returns the number of attached accelerator devices of the type specified by devtype. It must not be used inside parallel regions offloaded to an accelerator. Even though symbolic identifiers describing permitted devtype values may depend on the actual implementation, the standard recommends the following:

- acc_device_nvidia for Nvidia GPUs
- acc_device_radeon for AMD GPUs
- acc_device_xeonphi for Intel Xeon Phi processors.

```
acc_device_t acc_get_device_type();
```

This indicates the device type currently set as the target accelerator, and may return acc_device_none if the accelerator device has not been selected. Similar to acc_get_num_devices, it may not be called inside the accelerator region.

```
void acc_set_device_type(acc_device_t devtype);
```

This sets the type of device to be used as the accelerator for parallel regions of code. The device type is indicated by the input argument. Calling this function may result in undefined behavior (including program abort) if devices of the requested type are not available or the program was not compiled to support execution on the specified accelerator type. This function may not be called inside the accelerated region of code.

```
int acc_get_device_num(acc_device_t devtype);
```

The function returns the number (index) of the accelerator device of the specified type that will be used by the current thread to offload the parallel computations. As before, it may not be called inside the code region to be executed on the accelerator.

```
void acc_set_device_num(int n, acc_device_t devtype);
```

This defines which accelerator device of the specified type may be used to execute parallel regions by the current thread. If the value of n is negative, the implementation will select a default accelerator device. If devtype is zero, the specified number will be assumed for all attached accelerator types. Function execution may result in undefined behavior if n is greater than or equal to the number of

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 479

devices available of the indicated type. Again, acc_set_device_num may not be called from within the accelerated code region.

Example:

```
1  #include <stdio.h>
2  #include <openacc.h>
3
4  int main() {
5    printf("Supported OpenACC revision: %d.\n", _OPENACC);
6
7    int count = acc_get_num_devices(acc_device_nvidia);
8    printf("Found %d Nvidia GPUs.\n", count);
9    int n = acc_get_device_num(acc_device_nvidia);
10   printf("Default accelerator number is %d.\n", n);
11
12   count = acc_get_num_devices(acc_device_host);
13   printf("Found %d host processors.\n", count);
14   n = acc_get_device_num(acc_device_host);
15   printf("Default host processor number is %d.\n", n);
16 }
```

Code 16.1. Example code illustrating the use of the OpenACC library functions.

The example program shown in Code 16.1 invokes several library functions and has been compiled to run on a Cray XK7 system containing AMD Opteron CPUs and Nvidia Kepler GPUs. Launching it on a node equipped with a single GPU prints the following:

```
Supported OpenACC revision: 201306.
Found 1 Nvidia GPU(s).
Default accelerator number is 0.
Found 1 host processors.
Default host processor number is 0.
```

The retrieved release date is June 2013, which corresponds to OpenACC specifications revision 2.0. All the following code examples presented in this chapter were executed in the same environment.

16.4 OPENACC ENVIRONMENT VARIABLES

Currently, OpenACC defines only three environment variables that may be used to modify the runtime behavior of applications.

- ACC_DEVICE_TYPE determines the default device type which will be used to accelerate the marked parallel regions of the code. This value is implementation dependent. For example, the PGI compiler permits the values of NVIDIA, RADEON, and HOST to signify respectively the selection of an Nvidia or AMD branded GPU as the target accelerator device or execution on the host processor. The program has to be compiled in a way that enables the use of multiple accelerator devices.

- `num_gangs(`*integer-expression*`)`
  The `num_gangs` clause is used to specify explicitly the number of gangs across which the workload is distributed. If absent, an implementation-specific default is used. Note that restrictions imposed by the target architecture may cause the implementation to choose a lower number of gangs than that requested.
- `num_workers(`*integer-expression*`)`
  Analogous to `num_gangs`, this clause requests the specific number of workers per gang used for execution of the parallel workload in WP mode. The default number of workers is chosen if not specified, in which case it is not guaranteed to be consistent between different parallel regions (marked by the `parallel` or `kernel` directives) invoked by the program. As mentioned above, the particular implementation may modify the number of workers due to architectural constraints.
- `vector_length(`*integer-expression*`)`
  This requests the specific number of vector lanes to be assigned to each worker for code segments annotated by the `vector` clause with the `loop` directive (discussed later). Due to the arrangement of execution resources, the implementation is free to choose a value that better matches hardware specifications.

In addition to these, data management clauses may be present; these are discussed in Section 16.5.3.

Example:

```
1  #include <stdio.h>
2
3  const int N = 1000;
4
5  int main() {
6    int vec[N];
7    int cpu_sum = 0, gpu_sum = 0;
8
9    // initialization
10   for (int i = 0; i < N; i++) vec[i] = i+1;
11
12   #pragma acc parallel async
13   for (int i = 100; i < N; i++) gpu_sum += vec[i];
14
15   // the following code executes without waiting for GPU result
16   for (int i = 0; i < 100; i++) cpu_sum += vec[i];
17
18   // synchronize and verify results
19   #pragma acc wait
20   printf("Result: %d (expected: %d)\n", gpu_sum+cpu_sum, (N+1)*N/2);
21
22   return 0;
23 }
```

Code 16.2. Example of concurrent GPU and CPU execution triggered by the async clause.

The example application listed in Code 16.2 sums all components of a 1000-element vector. The first 100 elements are added on a CPU, while the GPU asynchronously sums the remaining 900 numbers at the same time. Synchronization with the GPU is achieved in line 19 preceding the result output. It uses a `wait` directive and not a `wait` clause on a `parallel` directive, since the latter would require an executable workload to be specified. That way, the `printf` statement immediately following in line 20 is executed by the host. The `parallel` directive allows the user to define precisely the way in which the affected workload is parallelized, but by default it is not going to parallelize anything (the execution is started in GR mode). As there are no additional parallelization clauses specified in line 12, the compute region in line 13 is not going to be vectorized. Since the code does not use any OpenACC library calls or macros, it is not necessary to include the OpenACC header file. The program produces the following output:

```
Result: 500500 (expected: 500500)
```

16.5.2 KERNELS CONSTRUCT

The compiler encountering the `kernels` directive performs the analysis of marked sections of the code and converts these into a sequence of parallel kernels that will be executed in order on the accelerator device. The number of gangs and workers and vector size may be different for each such kernel. The workload subdivision is typically performed in a way that creates one kernel for each loop nest present in the code. The primary difference between the `kernels` construct and the `parallel` directive is that the latter relies on the programmer to configure various parameters that divide the workload across accelerated execution resources. Thus the use of the `kernels` directive is recommended for beginners to OpenACC programming, but it may not always yield the best-performing code. Its syntax is shown below:

`#pragma acc kernels` *[clause-list]*

structured-block

The `kernels` construct accepts `async` and `wait` clauses that behave as described for the `parallel` clause, as well as data management clauses (discussed further in Section 16.5.3). Similar restrictions to those of the **parallel** directive apply: the code may not branch out or into the accelerated region.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 482

Example:

```
1  #include <stdio.h>
2
3  const int N = 500;
4
5  int main() {
6    // initialize triangular matrix
7    double m[N][N];
8    for (int i = 0; i < N; i++)
9      for (int j = 0; j < N; j++)
10       m[i][j] = (i > j)? 0: 1.0;
11
12   // initialize input vector to all ones
13   double v[N];
14   for (int i = 0; i < N; i++) v[i] = 1.0;
15
16   // initialize result vector
17   double b[N];
18   for (int i = 0; i < N; i++) b[i] = 0;
19
20   // multiply in parallel
21   #pragma acc kernels
22   for (int i = 0; i < N; i++)
23     for (int j = 0; j < N; j++)
24       b[i] += m[i][j]*v[j];
25
26   // verify result
27   double r = 0;
28   for (int i = 0; i < N; i++) r += b[i];
29   printf("Result: %f (expected %f)\n", r, (N+1)*N/2.0);
30 }
```

Code 16.3. Accelerated matrix—vector multiply using the `kernels` directive.

The program listed in Code 16.3 performs multiplication of a matrix and a vector, the dimensions of which are known at compile time and fixed. The accelerated region of code follows the `kernels` directive in line 22 and contains a loop nest: the outer loop iterates over matrix rows (index i) and the inner loop over the columns (index j). Unlike Code 16.2, the execution of the parallel region is synchronous (there is no `async` clause), meaning that the program will not proceed to result verification until the accelerated kernel computation is finished. The result of program execution is shown below:

```
Result: 125250.000000 (expected 125250.000000)
```

16.5.3 DATA MANAGEMENT

The resultant speed-up of an accelerated program strongly depends on the efficiency of data transfers between host and accelerator memories. In some cases, such as for AMD accelerated processing units, the accelerator shares the address space with the host processor. The overheads of communicating the data structures between the two components are minimal, as they are simply accomplished through pointer passing without any explicit data copies. If an accelerator needs to perform computation on certain elements of the data array, it only has to compute the resulting address of the data element based on the supplied pointer value, element index, and data type, and dereference it (fetch the desired element from memory), just as the host processor would. Unfortunately, many accelerator devices utilized in current supercomputing installations feature separate memory modules that necessitate explicit data transfers. Ideally, such transfers would be orchestrated without involving any unnecessary data or even entirely avoiding communication when not required. The first case is apparent when performing computation only on a subset of array or vector elements; copying the entire structure would only increase the latency data offload. The second scenario may arise when a dataset produced as a result of GPU computation would overwrite the contents of an array originally created on the host. Copying the initial state of such an array to the GPU before performing the accelerated computation is obviously unnecessary.

Unfortunately, due to the complexity of C and C++ code, static analysis of data access patterns by the compiler cannot always determine with certainty which portions of the affected data structures should be offloaded to the accelerator. OpenACC by default chooses correctness over efficiency and performs full bidirectional copies, i.e., transfer of the initial state of all involved data structures to the device before initiating accelerated computations and copying back the possibly updated state of involved datasets after the accelerated region's execution completes. Note that this is supported implicitly only when the dimensions of the involved arrays are known at compile time; for dynamically allocated arrays or arrays that are passed by pointer, it is a good idea to specify explicitly the ranges of data that should be offloaded to avoid potential out-of-bounds access errors during runtime. OpenACC implementations may further optimize (or even avoid) the data transfers if the accelerator is capable of accessing the host memory directly.

OpenACC provides the following clauses to control data copying between the host and accelerator memories.

- `copy`(*variable-list*)
  This makes data copies upon entry to and exit from the parallel region. First, for each variable specified in the variable list, the runtime system checks if the required data exists in the accelerator memory. If so, its reference count is incremented; otherwise a sufficient accelerator memory is allocated and a data copy from host memory to the allocated memory is arranged. The corresponding reference count for the data structure is set to one. On exit from the parallel region, the reference count is decremented. If it reaches zero, the corresponding data is copied back to the host memory and the allocated memory segment on the accelerator is deallocated.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 484

- `copyin(`*variable-list*`)`
  This makes data copies upon entry to the parallel region. It behaves as a one-directional version of the `copy` clause. All operations specified for region entry in the `copy` clause are executed without modification. However, on exit from the parallel region the reference counts for all data structures specified in the variable list are decremented. If the count for a specific variable reaches zero, the corresponding device memory is deallocated, but no data transfer to host memory takes place.
- `copyout(`*variable-list*`)`
  This makes data copies upon exit from a parallel region. The `copyout` clause may be viewed as a complement to the `copyin` clause. Upon entry to the parallel region, if the data are already present in the accelerator memory, their reference counter is incremented. If not, the sufficient memory segment is allocated in the device memory and the reference count for it is set to one. The allocated memory is not initialized (and no data transfer takes place).
  Upon exit, the reference count for the involved data structures is decremented. If it reaches zero, the data are copied back to the host memory and the corresponding memory segment on the device is deallocated.
- `create(`*variable-list*`)`
  This creates a data structure on the accelerator to be used by local computation. The `create` clause never transfers any data between the host and accelerator memories. When the affected parallel region is entered and the data structure already exists in the device memory, the runtime increments the reference counter; otherwise a suitable amount of device memory will be allocated, with the reference count set to one. On exit the reference count is decremented, and if it reaches zero the corresponding memory is deallocated.

The *variable-list* specifier accompanying the clauses listed above contains identifiers of program variables that are subjected to data copy operations. The identifiers are separated by a comma ("`,`"). They may be optionally followed by a range specification consisting of a pair of square brackets per dimension, each enclosing the index range specification. The index range consists of two integer expressions separated by a colon ("`:`"), with the first integer value denoting the starting index and the second value indicating the length (number of contiguous elements per dimension). If the first number is omitted, zero is assumed. The second number may be omitted if the size of the array is known at compile time, and implies that the full dimension is used. Thus `a[5:t]` describes the range of elements of vector `a` starting at index 5 and containing `t` elements, i.e., the sequence `a[5]`, `a[6]`, ..., `a[5+t-1]`. Analogously, `mat[:N][16:32]` refers to a rectangular segment of array `mat` that comprises 32-element-long fragments of its first N rows. Each such fragment starts at index 16. The entire dataset thus includes N × 32 array elements.

Thanks to compiler support, OpenACC supports several different ways in which arrays may be defined in C and C++ programs.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 485

1. Statically allocated arrays with fixed bounds, such as:

```
int cnt[4][500];
```

   One important restriction related to specifying the data transfer range for statically allocated arrays is that it must identify a contiguous chunk of memory. Only the range specifier for the first dimension may describe a subset of elements, while the specifiers for the remaining dimensions must identify full bounds. Thus for the declaration above `cnt[2:2][:500]` (last two rows of matrix `cnt`) is legal, whereas `cnt[:4][0:100]` (first 100 columns of matrix `cnt`) is not.

2. Pointers to fixed-bound arrays:

```
typedef double vec[1000];
vec *v1;
```

3. Statically allocated array of pointers:

```
float *farray[500];
```

4. Pointer to array of pointers:

```
double **dmat;
```

Multidimensional array definitions may include mixed declarations involving static bounds and pointers. To follow the range specification constraints correctly in a general case, it may be helpful to realize that the runtime system will mirror the organization of the source data structures from the host on the accelerator, allocating pointers where necessary and filling in their values. Once the data structures are defined, modification of the embedded pointers on the host or device is discouraged. To demonstrate the application of improved data management techniques to Code 16.3, it is rewritten to support dynamically allocated arrays storing the main matrix data and input and output vectors. The result is listed in Code 16.4.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 486

Example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv) {
5   unsigned N = 1024;
6   if (argc > 1) N = strtoul(argv[1], 0, 10);
7
8   // create triangular matrix
9   double **restrict m = malloc(N*sizeof(double *));
10  for (int i = 0; i < N; i++)
11  {
12    m[i] = malloc(N*sizeof(double));
13    for (int j = 0; j < N; j++)
14      m[i][j] = (i > j)? 0: 1.0;
15  }
16
17  // create vector filled with ones
18  double *restrict v = malloc(N*sizeof(double));
19  for (int i = 0; i < N; i++) v[i] = 1.0;
20
21  // create result vector
22  double *restrict b = malloc(N*sizeof(double));
23
24  // multiply in parallel
25  #pragma acc kernels copyin(m[:N][:N], v[:N]) copyout(b[:N])
26  for (int i = 0; i < N; i++)
27  {
28    b[i] = 0;
29    for (int j = 0; j < N; j++)
30      b[i] += m[i][j]*v[j];
31  }
32
33  // verify result
34  double r = 0;
35  for (int i = 0; i < N; i++) r += b[i];
36  printf("Result: %f (expected %f)\n", r, (N+1)*N/2.0);
37
38  return 0;
39 }
```

Code 16.4. Example OpenACC matrix–vector multiply with improved data transfers.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 487

The size of the involved arrays may be defined (within reason) on the command line. To preserve the double-index notation when accessing the elements of matrix m rather than flattening it to a vector, it has been declared as a pointer to a vector of pointers to dynamically allocated rows (this corresponds to scenario 4 described above). The pointers are declared with the `restrict` attribute telling the compiler that it should not expect pointer aliasing and potentially leading to a better optimized code. Since both the input matrix m and vector v are not modified by the computation, they are declared in the `copyin` clause. Vector b does not need to be initialized from the host memory, since its entire content is overwritten by computation. It is therefore declared as a `copyout` variable. Since the accelerator can easily zero out individual elements of b before accumulating partial dot product values into it, this part of the computation has been explicitly moved to the accelerated region. Running the program with argument 2000 yields:

```
Result: 2001000.000000 (expected 2001000.000000)
```

16.5.4 LOOP SCHEDULING

The `loop` directive is one of the fundamental OpenACC constructs responsible for identifying and fine-tuning the parallelization of accelerated workloads. It may be specified either as a separate directive:

```
#pragma acc loop [clause-list]
for (...)
```

Or as a clause combined with a parent `parallel` or `kernels` directive. In any case, it applies to the for-loop immediately following the clause or directive. The available loop control clauses include the following:

- `collapse(`*integer-expr*`)`
  This specifies how many nested loop levels indicated by the argument value are affected by the scheduling clauses present in the directive. Normally only the nearest loop following the directive is considered. The argument must evaluate to a positive integer.
- `gang`
- `gang(`*[*`num:`*] integer-expr [, integer-expr...]*`)`
- `gang(static:`*integer-expr*`)`
- `gang(static:*)`

  This distributes iterations of the affected loop(s) across gangs created by the parent `parallel` or `kernels` directive.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 488

When used with the `parallel` construct, the number of gangs is determined by the parent directive, hence only the static argument is permitted in one of the two forms listed above. It indicates the *chunk* size: a count of loop iterations that is used as a unit of workload assignment. Chunks are assigned to gangs in a round-robin fashion. If the last form of gang specification is used, chunk size is determined by implementation. It should be stressed that for correct results loop iterations must be data independent (except for the reduction clause described below), since the compiler is not going to perform the full code analysis, as when using the `kernels` directive.

If the `loop` clause is associated with the `kernels` construct, all forms are permitted with some restrictions. The first two variants may be specified only if `num_gangs` does not appear in the parent `kernels` construct. If used with a numeric argument, it specifies the number of gangs to be used for parallel execution of the loop. The meaning of the static argument is as described above for the `parallel` construct.

- `worker`
- `worker([num:]`*integer-expr*`)`
  This causes the loop iterations to be distributed across the workers in a gang. When used with the `parallel` construct, only the first form is allowed. It causes the gang to switch to WP execution. The loop iterations must be data independent. When the parent directive is `kernels`, the form with an argument may be used only if `num_workers` was not specified in the parent construct. The expression must evaluate to a positive integer that indicates the number of workers per gang to be used.
- `vector`
- `vector([length:]`*integer-expr*`)`
  This enables execution of loop iterations in vector or SIMD mode. The conditions of use are analogous to those of the `worker` clause, except that they apply to vector-level parallelism.
- `auto`
  This forces analysis of data dependencies in the loop to determine if it can be parallelized. It is implied in every `kernel` directive that does not contain the `independent` clause.
- `independent`
  This instructs the compiler to treat the loop iterations as data independent, thus enabling more possibilities for parallelization. It is implied for all `parallel` directives that do not specify `auto` clauses.
- `reduction(`*operator*`:`*variable*`[,`*variable*`...])`
  The reduction clause marks one or more of the specified variables as a participant in the reduction operation performed at the end of the loop. The variable may not be an array element or a structure member. The supported operators include +, *, `max`, `min`, &, |, &&, and || for sum, product, maximum, minimum, bitwise-and, bitwise-or, logical-and, and logical-or, respectively.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 489

Example:

```
1  #include <stdio.h>
2
3  const int N = 10000;
4
5  int main() {
6    double x[N], y[N];
7    double a = 2.0, r = 0.0;
8
9    #pragma acc kernels
10   {
11     // initialize the vectors
12     #pragma acc loop gang worker
13     for (int i = 0; i < N; i++) {
14       x[i] = 1.0;
15       y[i] = -1.0;
16     }
17
18     // perform computation
19     #pragma acc loop independent reduction(+:r)
20     for (int i = 0; i < N; i++) {
21       y[i] = a*x[i]+y[i];
22       r += y[i];
23     }
24   }
25
26   // print result
27   printf("Result: %f (expected %f)\n", r, (float)N);
28
29   return 0;
30 }
```

Code 16.5. Example program using the `loop` directive with parallelism and reduction clauses.

The program listed in Code 16.5 showcases the use of the `loop` directive to perform accelerated vector scaling and accumulation reminiscent of the *daxpy* routine from the linear algebra package. For demonstration purposes, the initialization code has also been moved to the accelerator. It requests parallelization in WP mode with the default number of gangs and workers. The parallelization parameters of the computational loop are left to the discretion of the implementation. The loop is explicitly marked as data independent to promote this and avoid the compiler analysis which would be performed by default for the `kernels` construct (less sophisticated compilers may interpret the update

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 490

of `y[i]` as data dependence). To verify the correctness of the result, a reduction clause is used that sums all elements of the result vector `y` into variable `r`. The generated output is given below:

```
Result: 10000.000000 (expected 10000.000000)
```

16.5.5 VARIABLE SCOPE

It should be apparent at this point that the OpenACC treatment of variables participating in the computation varies depending if they are loop indices or data structures and where they are declared in the code. Loop variables are considered private to each thread that executes loop iterations. Variables declared in a block of code that is marked for execution in VP mode are private to the thread that is associated with each vector lane. For code executed in WP vector-single mode the variables are private to each worker, but shared across vector lanes associated with that worker. Similarly, variables declared in a block marked for worker-single mode are private to the containing gang, but shared across the threads operating at worker and vector levels in that gang.

OpenACC defines a `private` clause that may be used to restrict the sharing of variables further. It may be declared alongside the `parallel` or `loop` directive, and accepts a list of variable names as argument. In the first case, a copy of each variable in the list is generated for each parallel gang. In the loop context, a copy of each variable is created for each thread associated with each vector lane (VP mode). In vector-single WP mode, a copy of every item in the list will be created and shared for each set of threads associated with vector lanes in each worker. Otherwise, a variable copy is created and shared across all vector lanes of every worker in each gang. A `firstprivate` variant of the `private` clause is also available for the `parallel` directive with the same access semantics, except the variable copies are additionally initialized to the value of the variables inherent to the first thread encountering the `parallel` construct during the code execution.

16.5.6 ATOMICS

Parallelization of code across multiple execution resources on occasion calls for synchronization of access to some data structures that should be carried out in predefined order. This is enforced by the `atomic` construct with the syntax described below:

```
#pragma acc atomic [atomic-clause]
    statement;
```

Supported atomic clauses include `read`, `write`, `update`, and `capture`, depending on the type of access synchronization. If a clause is absent, an `update` clause is assumed. The `read` clause is used to force atomic access to variables on the right-hand side of the equal sign in an assignment statement. Analogously, the `write` clause protects writes to variables on the left-hand side of the equal sign in assignments. The `update` clause enforces correct updates of values of variables that have to be performed using read—modify—write sequence of operations. Examples include prefix and postfix increment and decrement operators as well as updates in the form of *op*=, where *op* is a binary operator such as +, -, *, etc. The `capture` clause refers to assignment statements in which the right-hand side is an atomic update expression such as described for the `update` clause, while the left-hand side is a variable supposed to capture the original or final value of the atomically modified variable (depending on the operation type).

Example:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv) {
4    if (argc == 1) {
5      fprintf(stderr, "Error: file argument neede!\n");
6      exit(1);
7    }
8    FILE *f = fopen(argv[1], "r");
9    if (!f) {
10     fprintf(stderr, "Error: could not open file \"%s\"\n", argv[1]);
11     exit(1);
12   }
13
14   const int BUFSIZE = 65536;
15   char buf[BUFSIZE], ch;
16   // initialize histogram array
17   int hist[256], most = -1;
18   for (int i = 0; i < 256; i++) hist[i] = 0;
19
20   // compute histogram
21   while (1) {
22     size_t size = fread(buf, 1, BUFSIZE, f);
23     if (size <= 0) break;
24     #pragma acc parallel loop copyin(buf[:size])
25       for (int i = 0; i < size; i++) {
26         int v = buf[i];
27         #pragma acc atomic
28         hist[v]++;
29     }
30   }
31   // print the first highest peak
32   for (int i = 0; i < 256; i++)
33     if (hist[i] > most) {
34       most = hist[i]; ch = i;
35   }
36   printf("Highest count of %d for character code %d\n", most, ch);
37
38   return 0;
39 }
```

Code 16.6. Example program showing the application of the `atomic` clause.

The program presented in Code 16.6 calculates a histogram of ASCII character occurrences in a file given as the command-line argument. The `atomic` directive in line 27 (implied `update` clause) ensures the correct increment of the histogram bin for a specific character. Running the code for file containing the first paragraph of the "*lorem ipsum*" text [10] produces:

```
Highest count of 68 for character code 32
```

16.6 SUMMARY AND OUTCOMES OF CHAPTER 16

- There are several programming environments for accelerators; they differ in approach, scope, supported features, and availability. The most commonly used include CUDA, OpenCL, OpenACC, and C++ AMP.
- OpenACC is a GPU and accelerator programming framework that attempts to simplify parallel programming and achieve better programmability by using a directive-based approach similar to OpenMP. It requires a specialized compiler capable of generating executable accelerator code following the static analysis of appropriately marked source code. Compilers with OpenACC support are available from PGI, Cray, and several open-source communities (OpenUH, OpenARC, and GCC).
- The main method of identifying potential parallel execution regions is through the addition of suitable "`#pragma acc`" directives in the relevant places in source code. In addition to directives, the execution of programs is affected by predefined library calls and environment variables.
- OpenACC programs rely on the host machine to initiate the program computations and offload the data and executable code to the accelerator at appropriate times. Accelerated code execution is by default synchronized with the execution of the nonaccelerated sections of the program on the host machine. Additional speed-up may be obtained by asynchronously coscheduling computations on the GPU with computations on the host processor.
- Performance gains in regions executed on the accelerator are realized through parallelization at three levels: gang, worker, and vector (from the coarsest to the finest grain). The programmer retains control of parameters influencing each level, although he/she may also select implementation defaults.
- There are two main compute directives: "`parallel`" and "`kernels`." The first forgoes much of the correctness analysis of the source code, relying on the programmer to verify data independence between concurrently executing accelerator threads. The second performs a thorough static analysis of the code, and allows vectorization and parallel execution only if it is safe to do so.
- Distribution of regular and nested loop iterations across the accelerated execution resources is one of the primary methods of increasing application performance gains. It is controlled by the `loop` clause, which also supports an accelerated set of reduction operations.
- Overall application performance depends on the efficiency of data transfers between accelerator and host memories. OpenACC supports additional control clauses to optimize this aspect of execution (`copy`, `copyin`, `copyout`, `create`).
- OpenACC provides simple mechanisms for synchronization of access to critical variables from multiple accelerator threads to ensure the correctness of program execution. Four modes of atomic access are supported: `read`, `write`, `update`, and `capture`.

16.7 QUESTIONS AND PROBLEMS

1. Characterize directive-based programming. How does it differ from using functionality provided by software libraries?

2. Write an OpenACC program to compute the approximation of the natural logarithm of 2 using the first 10,000,000 terms of Maclaurin expansion:

$$\ln(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{4}x^4 + \ldots$$

Make sure the generated accelerator code is parallelized.

3. Modify Code 16.6 to compute the frequency of alphabetic digraph (two-letter sequence) occurrence in a block of text. Ignore case sensitivity.
4. Write a simple OpenACC program that computes the average value of elements occupying the lower triangular part (i.e., all elements on and below the main diagonal) of a large square matrix. Is it possible to optimize the program so that:
   a. efficiency of data transfers is improved (by avoiding copying data not used by computation)?
   b. the work performed in each iteration is balanced across GPU threads?
   Implement optimizations that are possible. How do they affect performance? Test several different matrix sizes.
5. To debug an OpenACC program, the irrelevant portions of the code were removed, yielding the following:

```
 1 #include <stdio.h>
 2
 3 const int N = 100, M = 200;
 4
 5 int main() {
 6   int m[N][M];
 7   for (int i = 0; i < N; i++)
 8     for (int j = 0; j < M; j++)
 9       m[i][j] = 1;
10
11   #pragma acc kernels
12   for (int i = 0; i < N; i++)
13     for (int j = M-i; j < M; j++)
14       m[i][j] = i+j+1;
15
16   // verify result
17   int errcnt = 0;
18   for (int i = 0; i < N; i++)
19     for (int j = 0; j < M; j++) {
20       int expect = (j >= M-i)? i+j+1: 1;
21       if (m[i][j] != expect) errcnt++;
22     }
23   printf("Encountered %d errors\n", errcnt);
24   return errcnt != 0;
25 }
```

The code fails (produces a nonzero error count) when compiled with certain OpenACC compilers. What may be the reason for that? How may the errors be prevented?

REFERENCES

[1] Khronos Group, OpenGL: The Industry's Foundation for High Performance Graphics; Version 4.5 Specifications, Khronos Group, 2016 [Online]. Available: https://www.opengl.org/documentation/current_version/.
[2] Microsoft Corporation, Getting Started with DirectX Graphics, 2016 [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/hh309467.
[3] E.S. Larsen, D. McAlister, Fast matrix multiplies using graphics hardware, in: Proceedings of Supercomputing 2001, 2001.
[4] Nvidia Corporation, CUDA Toolkit Documentation v8.0, September 27, 2016 [Online]. Available: http://docs.nvidia.com/cuda/.
[5] Khronos Group, The OpenCL Specification (provisional), Version 2.2, March 11, 2016 [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf.
[6] Microsoft Corporation, C++ AMP: Language and Programming Model, v1.2, December, 2013 [Online]. Available: http://download.microsoft.com/download/2/2/9/22972859-15C2-4D96-97AE-93344241D56C/CppAMPOpenSpecificationV12.pdf.
[7] The OpenACC Application Programming Interface, Version 2.5, OpenACC-Standard.org, October, 2015 [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf.
[8] OpenUH – Open Source UH Compiler (Source Repository), 2015 [Online]. Available: https://github.com/uhhpctools/openuh.
[9] S. Lee, J. Vetter, OpenARC: extensible OpenACC compiler framework for directive-based accelerator programming study, in: WACCPD: Workshop on Accelerator Programming Using Directives in Conjunction with SC'14, 2014.
[10] Lorem Ipsum Generator, [Online]. Available: http://www.lipsum.com.

CHAPTER

MASS STORAGE

17

CHAPTER OUTLINE

17.1 Introduction **509**
17.2 Brief History of Storage **512**
17.3 Storage Device Technology **514**
17.3.1 Hard Disk Drives 514
17.3.2 Solid-State Drive Storage 520
17.3.3 Magnetic Tape 524
17.3.4 Optical Storage 529
17.4 Aggregated Storage **534**
17.4.1 Redundant Array of Independent Disks 534
17.4.1.1 RAID 0: Striping 534
17.4.1.2 RAID 1: Mirroring 535
17.4.1.3 RAID 2: Bit-Level Striping With Hamming Code 536
17.4.1.4 RAID 3: Byte-Level Striping With Dedicated Parity 536
17.4.1.5 RAID 4: Block-Level Striping With Dedicated Parity 537
17.4.1.6 RAID 5: Block-Level Striping With Single Distributed Parity 538
17.4.1.7 RAID 6: Block-Level Striping With Dual Distributed Parity 539
17.4.1.8 Hybrid RAID Variants 539
17.4.2 Storage Area Networks 541
17.4.3 Network Attached Storage 543
17.4.4 Tertiary Storage 544
17.5 Summary and Outcomes of Chapter 17 .... **545**
17.6 Questions and Problems **546**
References **547**

17.1 INTRODUCTION

The storage subsystem is one of the key components of every computing platform. Although the organization, speed, capacity, and supported functions of storage vary depending on platform class, its presence is always required for computations to be carried out. In high performance computing (HPC) one can observe quite possibly the broadest variety of storage options and involved storage technologies as well as range of implementation scales. This chapter discusses the segment of storage

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00017-4
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 496

technology and low-level techniques utilized to support the requirements of HPC systems reliably to preserve the high volume of computational state in the form of both scientific data and elements of the operating environment. The state retention must be persistent between the power cycles of the machine for it to be able to execute bootstrap procedures on restart, attain the correct operational status, and resume interrupted computational tasks. This part of the storage hierarchy is referred to as "mass storage" to reflect its capability to absorb large amounts of data. Mass storage is not concerned with volatile devices, such as main memory or processor registers. Besides input and output (I/O) datasets used by and produced as a result of computation, mass storage preserves the code (executables and libraries) necessary to run the operating system and its associated background management processes, configuration, and update scripts, as well as the user's and system administrator's tools and utilities. Finally, mass storage plays an integral role in checkpoint and restart of compute applications, alleviating the impact of temporal and system resource limits imposed on application execution.

Traditionally, the storage hierarchy is subdivided into four levels that differ in access latency and supported data bandwidth, with latencies increasing and effective transfer bandwidth dropping when moving away from the top level of the hierarchy. At the same time, storage capacity rapidly grows. The commonly recognized hierarchy levels are as follows.

- *Primary storage*, which comprises system memories, caches, and CPU register sets. This type of storage is predominantly volatile (loses data contents when powered off), with the exception of read-only memories (ROMs) that store firmware or CPU boot code. While there have been some efforts to utilize various types of nonvolatile random access memories (NVRAMs) as a part of the overall memory pool accessible to processors, their access latencies typically prohibit achieving good integration, requiring dedicated and nontransparent support from the operating system (OS) and applications. The data access latencies range from a single CPU clock cycle (a fraction of a nanosecond) for registers to several hundred cycles for dynamic memories in remote non-uniform memory access domains; the respective bandwidths span from over 100 GB/s (SIMD registers in a single core) down to a few GB/s per bank of double data rate memory (such as DDR3, still in use in many installations). Aggregate memory size in HPC ranges from a few tens of gigabytes for small nodes to hundreds of gigabytes for nodes dedicated to memory-intensive tasks.
- *Secondary storage* is the first level of storage that leverages mass-storage devices. Normally CPUs cannot directly access the secondary (or higher-level) storage and therefore transfers of data between primary and secondary storage have to be mediated by the OS and computer chipset. The granularity of data access is typically limited to fixed-size blocks, while most primary storage devices operate at byte resolution. The most commonly used technology in this tier are hard disk drives (HDDs), which offer the industry's best cost per unit of storage coupled with satisfactory reliability. Over the last decade, however, their dominance in the market has been slowly eroding due to the introduction of high-capacity solid-state storage. The random access latency of secondary storage media may be less than 100 μs for the fastest solid-state devices to as much as tens of milliseconds for HDDs. The bandwidths may range from just below 100 MB/s for slower HDDs to a single GB per second for solid-state devices. HDDs still maintain the lead in total capacity, with up to 10 TB per single device.
- *Tertiary storage* is distinguished from secondary storage in that it usually involves large collections of storage media or storage devices which are nominally in an inaccessible or powered-off state, but may be reasonably quickly enabled for online use. Activation is typically

accomplished by automated mechanisms such as robots that physically move the requested mass-storage medium from its assigned long-term retention slot to the specified online access device (drive). To lower contention between multiple users, tertiary storage equipment typically hosts several independent media drives that may be accessed concurrently. Examples of tertiary storage equipment include tape libraries and optical jukeboxes. Since the bandwidth of a single drive is often insufficient to sustain many concurrent I/O requests, the content of the selected medium is copied to secondary storage first (e.g., disk cache). The access latency to tertiary storage may be substantially greater than that of secondary storage, especially when multiple competing requests must be serviced. In a contention-free state it typically takes single tens of seconds for the robot to grab and mount the medium, and the achieved single-device bandwidths are comparable to those of secondary storage. The storage capacity of robotic jukeboxes may reach as much as multiple hundreds of petabytes.

- *Offline storage* requires human intervention to enable access to the storage medium. It is primarily employed to archive, frequently in a secure location off site, precious information. Since the storage unit is not under the direct control of any computer, this provides a much-needed "air gap" to protect the security, confidentiality, and integrity of the archives. Offline storage is in principle similar to tertiary storage, although lack of predictability related to medium load requests results in highly random latency figures and it may not be considered a practical high performance solution other than for some niche applications.

The design and deployment of supercomputing storage subsystems comes with their own set of challenges. The prevailing trends of the past few decades have shown steady increases not only in memory capacity due to Moore's law, but also in supercomputing platform scale expressed as number of nodes per machine. As the aggregate size of computed datasets is roughly proportional to the total system memory size, this has resulted in a superlinear increase in demand for mass-storage capacity. Moreover, each successive generation of dynamic random access memory (DRAM) improved data transfer bandwidth, thus enabling faster data creation rates. At the same time, I/O device bandwidth exhibited comparatively modest growth and over the last decade effectively leveled out. Storage capacity per device originally loosely followed Moore's curve, but suffered from highly limited growth rates throughout most of the 2010s. This resulted in a continuously increasing storage performance gap, and the time required to save or retrieve the data occupying a significant fraction of a machine's memory is rising as well. In extreme cases checkpoint or restart of large applications may take several hours.

The addition of global high-bandwidth networks, such as Internet2 [1], has enabled access to collections of data at remote sites as well as input data streaming. In many cases the expansion of the input dataset is reflected by the volume of generated output and/or intermediate data, additionally stressing the local storage subsystem. This is particularly relevant to a relatively new class of data-intensive applications collectively known as "Big Data" which, in addition to operating on large data volumes and requiring substantial processing speeds, are frequently hampered by intrinsic variety and irregularities of the processed data structures. As the storage capacity scales linearly with the number of I/O devices, support of large volumes of data results in I/O subsystems occupying a significant amount of floor space at data centers and drawing substantial amounts of electric power. Since the bulk of secondary storage capacity is provided by electromechanical devices such as disk drives, the data centers must install measures to deal with common device failures. Even though devoid of moving parts, solid-state storage devices are not immune to failures either, and these are exacerbated by dissipated heat and the number of data rewrites per device. To maintain the operation the centers must therefore provide redundant storage, further expanding the system's volume and energy requirements.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 498

Efficient data transfer between the primary and other storage levels requires significant dedicated interconnect bandwidth. Unfortunately, large machine procurement practices at many institutions frequently focus on components directly related to computations, such as processors, memory, and network. Storage considerations are often secondary and based on poor analysis of requirements. This produces bandwidth-starved implementations with insufficient reliability and performance that, in some cases, share the I/O load with other components of the system (such as login nodes). While an increase in network switch capacity to provide the required bandwidth from compute nodes to mass storage may visibly impact the final system's cost, it will yield a better-balanced computing platform.

The challenges outlined above apply to many systems currently in service. While there is no single universal solution to address them, their impact may be alleviated through exploration of better I/O architectures, hardware-level solutions, and advances in the software stack. Architectural solutions may introduce additional intermediate hierarchy levels that provide high performance data sinks and sources in close vicinity to compute nodes. Such storage devices are capable of high-bandwidth communication with the nodes to satisfy the most urgent I/O requests with low latency, while constantly performing in the background slower data exchanges with larger storage devices located lower in the hierarchy. An example of this is the Cray burst buffer technology [2], which provides a number of nodes equipped with fast solid-state storage and regularly interspersed with other compute nodes. The burst buffer nodes have the benefit of the full Aries interconnect [3] bandwidth, but can also use a fraction of switch performance to interact with the storage servers. Hardware improvements are primarily focused on building more reliable, faster, and higher-capacity mass-storage devices. This is expected to lower power consumption, reduce the volume occupied by the secondary storage subsystem, and decrease the costs of ownership by requiring fewer spare storage devices to replace those that fail. An overview of these advances is discussed in the remainder of this chapter. Finally, software solutions arising from the design of better storage abstractions that embrace parallelism and asynchrony of access (such as the parallel file systems described in Chapter 18) can anticipate the I/O access patterns utilized by applications, fetch the required data ahead of time, and forward it to the memory of the prospective client, or provide smart checkpoint and restart that can gracefully overlap compute state management (saving, retrieval, transformation, compression) with ongoing computations. Software improvements may also directly address deficiencies or extend functionality of specific components in the system. For example, locating the data preprocessing and postprocessing engine on a storage node may conserve the network bandwidth required to ship the data between storage devices and compute nodes.

17.2 BRIEF HISTORY OF STORAGE

Technological progress brought dramatic improvements in both capacity and performance of mass-storage devices over the course of several decades. As illustrated in Fig. 17.1, starting with punched cards as the first external information store in the mid-1940s, through tape drives in the early 1950s, and continuing with HDDs from the mid-1950s to the present day, storage capacity grew an amazing 11 orders of magnitude. The increases in device storage capacity were reflected by the corresponding improvements in device I/O bandwidth (Fig. 17.2), which advanced six orders of magnitude over the same period. However, access latency improvements were far more modest, decreasing from single and tens of seconds for punched cards and tape to a few milliseconds in modern HDDs. Latency still remains one of the biggest performance bottlenecks plaguing most of the I/O devices in use today.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 499

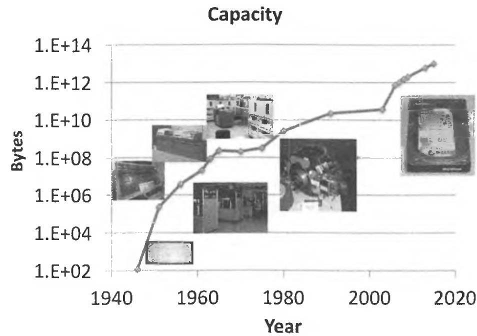


FIGURE 17.1

Increases in mass-storage capacity. Represented systems are a punch card on the ENIAC (1946), a Uniservo tape drive (1951), IBM 350 (1956), IBM 1301 (1961), IBM 1302 (1963), IBM 2314 (1965), IBM 3330 (1970), IBM 3350 (1975), IBM 3380 (1980), IBM 3390 (1991), Western Digital Raptor (2003), Seagate Barracuda 7200.10 (2006), HGST Deskstar 7K1000 (2007), Seagate Barracuda 7200.11 (2008), Western Digital WD20EADS (2009), HGST Ultrastar He6 (2013), and HGST Ultrastar He10 (2015).

Punchcard, UNIVAC I, and IBM 3380 photos by Arnold Reinhold via Wikimedia Commons. IBM 305 photo by US Army Red River Arsenal via Wikimedia Commons. IBM 2314 photo by Scott Gerstenberger via Wikimedia Commons

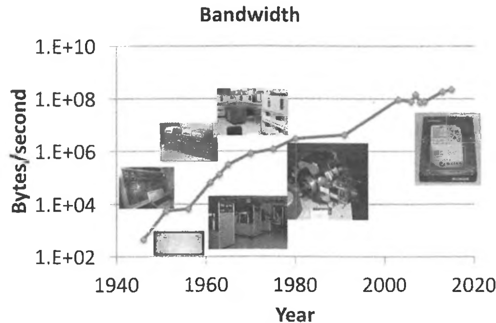


FIGURE 17.2

Improvements in I/O data access bandwidth. Represented systems include a punch card on the ENIAC (1946), a UNISERVO tape drive (1951), IBM 350 (1956), IBM 1301 (1961), IBM 1302 (1963), IBM 2314 (1965), IBM 3330 (1970), IBM 3350 (1975), IBM 3380 (1980), IBM 3390 (1991), Western Digital Raptor (2003), Seagate Barracuda 7200.10 (2006), HGST Deskstar 7K1000 (2007), Seagate Barracuda 7200.11 (2008), Western Digital WD20EADS (2009), HGST Ultrastar He6 (2013), HGSTUltrastar He10 (2015).

Punchcard, UNIVAC I, and IBM 3380 photos by Arnold Reinhold via Wikimedia Commons. IBM 305 photo by US Army Red River Arsenal via Wikimedia Commons. IBM 2314 photo by Scott Gerstenberger via Wikimedia Commons

17.3 STORAGE DEVICE TECHNOLOGY

As illustrated by the preceding section, the technology of hardware storage devices continuously evolved to support the ever-increasing demands for storage capacity and data access bandwidth. Currently the majority of storage systems utilize four main types of mass storage devices: HDDs, solid-state drives (SSDs), magnetic tapes, and optical storage. Although they serve largely the same purpose, they substantially differ in the underlying physical phenomena used to implement data retention as well their operational characteristics and cost. The fundamental properties and working principles of modern storage devices are discussed below.

17.3.1 HARD DISK DRIVES

HDDs have a long history as a data storage device in computing. Introduced in 1956, the first hard drive used in the IBM 350 RAMAC system [4] was approximately 68″ high, 60″ deep, and 29″ wide, and weighed approximately one ton. It contained 50 platters (disks serving as the recording medium for data) with a diameter of 24 in rotating at 1200 revolutions per minute (RPM). It stored 5 million six-bit characters that were transferred at a rate of 8800 per second. The successor drives appearing in the 1960s featured removable platter packs that could be moved between the different drive enclosures. Many improvements utilized by modern HDDs were developed in that decade, such as a multiple read—write head assembly that avoided the delay of head movement from one data platter to another, aerodynamic head design that permitted stable head operation in very close proximity to the recording medium, and the first voice—coil actuator. The introduction of the "Winchester" design in the early 1970s, using a dedicated portion of the media as a landing zone for read—write heads, marked the return to nonswappable platters (hence the occasionally used alternative name "fixed-disk drive"). The rotary actuator, a common component of modern HDDs, was developed by IBM in 1974 and used in its Gulliver [5] line of drives. The first disk drive approximating form factors broadly used today was released by Shugart Technology (now Seagate) in 1980; it featured 5.25″ housing, stored 5 MB of data, required an external controller board, and could be mounted inside larger personal computers such as the IBM PC [6]. Ongoing developments in this decade brought the familiar 1″ high 3.5″ (Conner Peripherals CP3022 storing 21 MB) and 2.5″ (PrairieTek 220 with a capacity of 20 MB) form factors. The 1990s brought many improvements in drive speed and capacity prompted by the development of partial response maximum likelihood (PRML) technology [7] (see below) for reliable decoding of weak signals retrieved from media and successive application of the giant magnetoresistive (GMR) [8] phenomenon to disk head design. Progress in storage areal density increase enabled a 1.8″ drive to be created in 1991 (Integral Peripherals 1820 with over 20 MB per disk), followed by IBM's 1″ Microdrive in 1999 that stored 340 MB of data. As the flash memory technology could not support competitive bit densities in that period, such miniature HDDs from multiple manufacturers were used as content storage for portable media players, among others the Apple iPod. At the same time, Seagate's Cheetah drives became one of the first to feature the record-breaking 10,000 RPM and later 15,000 RPM spindle speeds. Advances that followed after the year 2000 leveraged perpendicular magnetic recording to increase information density further on storage media, continuously increased embedded buffer memory size to permit better latency management, shifted to glass-based platter substrates, introduced helium as a cavity-filling gas to minimize energy loses due to rotating platter drag and turbulence, and used shingled magnetic recording. This continued technological progress has

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 501

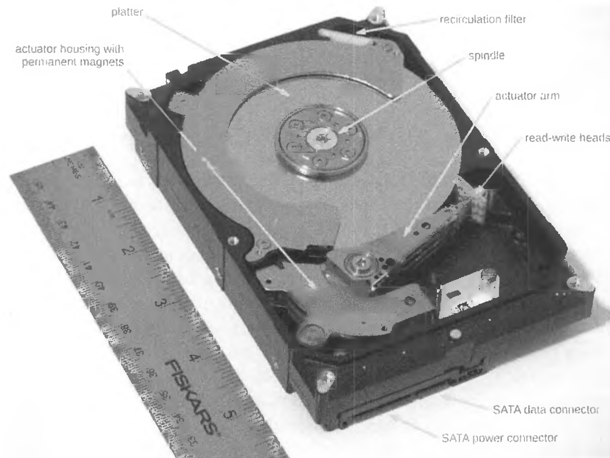


FIGURE 17.3

Internal components of a hard disk drive (2 TB Seagate HDD).

resulted in hard drives being able to store more information per device, provide faster access to data, consume less energy per operation, and last longer in a production environment.

Modern HDDs are a marvel of materials, electrical, and mechanical engineering. Their principal internal components are annotated in Fig. 17.3. The information is recorded on one or both surfaces of a disc-shaped *platter*. While the base material for platters is typically glass due to several well-mastered technological processes that guarantee the maximum surface flatness, the platters may also be made of aluminum or ceramics. The platters are polished to a roughness of less than 1 Å (10^{-10} m) and covered with several thin (single nanometers) layers of various materials containing cobalt, iron, nickel, ruthenium, platinum, chromium, and their alloys that promote the formation of the required crystallographic structure with properly oriented magnetic domains. The resulting material exhibits high coercivity, which is the ability to retain the acquired magnetization in the presence of an external magnetic field. The deposition of individual layers is done using a process called magnetron sputtering. The platter also receives a protective carbon-based coating through ion-beam or plasma-enhanced vapor deposition. Finally, a lubricant coat is deposited on the active surfaces and bonded. Storage densities of media manufactured this way exceed 800 Gb per square inch. A typical HDD stacks several platters on the same axle (spindle) to achieve the desired total storage capacity. The spindle is a part of a direct-drive brushless motor that rotates at several thousand RPM (commonly used speeds are 3600, 4200, 5400, 7200, and occasionally 10,000 and 15,000 RPM). Data are retrieved from and written to the platters using multiple read—write heads mounted at the end of the actuator arm. The arm can move in an arc over the platters to be able to locate a specific data track; the information is stored on platters in the form of concentric circles, referred to as *cylinders* to emphasize

the three-dimensional aspect of the data layout. The actuator motion is controlled by the so-called *voice coil*, named in analogy to a dynamic loudspeaker construction which has coils surrounded by permanent magnets which push the sound-generating membrane. Both mechanisms work due to Lorentz force causing the motion of a conductor in a magnetic field when electric current flows through it. While earlier implementations used stepper motors to move the heads, voice coils are a much more lightweight alternative and thus may achieve significantly faster movement at a lower energy profile.

Read–write heads are not attached directly to the actuator arms, but to *sliders*—tiny (that is, a fraction of millimeter in the longest dimension and weighing a fraction of a gram) aerodynamically shaped carriers that are responsible for maintaining the correct distance between the head and the spinning medium. Interestingly, no electrically powered techniques are used to stabilize the separation distance. Sliders are mounted on a gimbal assembly attached to the arm, and thus have some freedom of motion. Since the spinning platters force the boundary layer of air to move with them, this generates an aerodynamic force acting on the slider. The slider's surface consists of a number of patterns that generate both an air bearing with positive air pressure that pushes the slider away from the medium and a negative pressure area that pulls the slider closer to the surface. Since the relative linear motion of the slider with respect to the platter surface changes significantly for the inner and outer cylinders, the parameters of the slider's shape must be precisely calculated to provide nearly constant flight height in these conditions. In modern HDDs this distance is on the order of few nanometers.

Due to the precision involved, it is not difficult to see that foreign contaminants present a serious damage risk to HDDs. Most drives have ventilation outlets protected by additional filters to stop foreign matter. Some HDD versions are hermetically sealed and use inert gases such as nitrogen or helium to support their operation. Since debris may also be generated by nonfatal impacts of the slider with the medium, there is an additional built-in recirculation filter to contain the particulate matter. This works due to constant motion of the air propelled by the spinning platters.

Modern hard drives utilize multiple technologies to improve their access speeds and increase storage density. One breakthrough was the practical application of the GMR effect to the construction of read–write heads. A GMR head sandwiches a spacer of nonmagnetic metal between two layers of magnetic metal and adds a fourth antiferromagnetic layer to "pin" the magnetic orientation of the nearest magnetic layer. This structure, called a *spin valve*, demonstrates high sensitivity to weak magnetic fields (such as those recorded on the HDD medium) of the unpinned layer, resulting in substantial resistance changes following those of the external magnetic field. Besides information access, signals derived from the GMR head serve as a feedback to head movement servos, resulting in accurate positioning on top of the recorded track. Another critical technique is perpendicular recording, illustrated in Fig. 17.4. Due to the fundamental limit on magnetic domain size caused by the superparamagnetic effect, the traditional horizontal arrangement of domains results in poor utilization of the medium surface. Reorienting the domains vertically, which requires a specially formed recording medium via the multistage process mentioned above as well as the modification of the writing head's shape, produces increased bit density.

The peak media transfer speeds of current HDDs are in the order of 100–250 MB/s. In addition to user data, the recorded information contains error-correcting codes (ECCs) to detect and if possible correct any malformed data. The information in each track is subdivided into a number of sectors of constant size, each requiring an identifier, synchronization information, and an explicit gap separating it from its nearest neighbors. The standard for several decades was 512-byte sectors, but modern large-

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 503

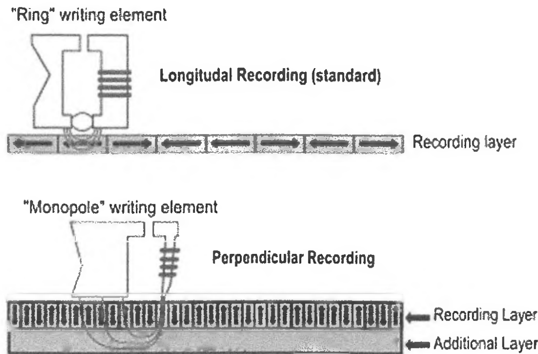


FIGURE 17.4

Bit density increase with perpendicular recording.

Diagram by Luca Cassioli via Wikimedia Commons, 2005

capacity drives forced manufacturers to migrate to 4096-byte sectors (called Advanced Format) to lower the spatial overheads of metadata, primarily ECCs, associated with each sector (see Fig. 17.5A). Older disks maintained a fixed number of sectors in each cylinder, hence producing a nonuniform recording density between the innermost and outermost tracks. Since the platters spin mostly at a constant rate, the solution was to introduce zone bit recording, illustrated in Fig. 17.5B. The platter surface is subdivided into concentric zones with different radii. Each zone features a specific number of sectors per track, thus allowing an increased number of sectors to be stored in the outer cylinders.

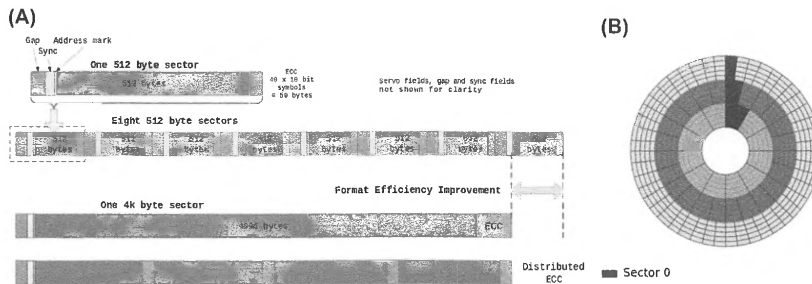


FIGURE 17.5

Physical information layout on HDDs: (A) advantage of larger sectors, (B) zone bit recording.

Diagrams by Dmitry Nosachev and Jan Schaumann via Wikimedia Commons

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 504

The continuing increases in bit density resulted in a smaller effective size of "bit area" and therefore weaker signals that still must be reliably detected. PRML is a signal processing technique responsible for boosting storage densities by as much as 40% while retaining a very high probability of correctly reconstructing the recorded information. In contrast to older methods relying on peak detection in read signal (which corresponds to points where the read head passes over domains, changing orientation of their magnetic field), PRML operates not only with weaker signals but signals where narrowly spaced domains may affect each other's magnetic field magnitude. The induced signals usually have too low an amplitude to register correctly with conventional peak detectors. PRML implementation consists of a variable-gain amplifier, an analog-to-digital converter, analog and digital filters, a clock recovery circuit, and finally a Viterbi [9] decoder running in real time analyzing serial input data streams at a rate of several gigabits per second. PRML inspired even more complex algorithms of signal reconstruction, such as the noise-predictive maximum likelihood [10] method.

Despite all the precautions, internal material imperfections and external shocks may eventually cause damage or otherwise degrade sections of recorded media. HDDs are manufactured with spare capacity that permits transparent remapping of damaged sectors. The only indication that this has happened is decreased sequential access performance; the sectors that were occupying the same physical track and could be read back-to-back during a single rotation of the platter may require additional head movement if they were migrated to different areas of the disk. Most drives available on the market are equipped with self-monitoring, analysis, and reporting technology (SMART) [11] that continuously monitors the health status of the device and may even warn the user ahead of time about an impending failure. While the interpretation details of individual values may be vendor specific, commonly reported parameters include start/stop count, spin-up time, seek and read error rates, total power-on hours, power cycle count, reallocated sector count, unrecoverable error count, command timeouts, current and highest recorded temperature, registered shock values, servo off-track errors, uncorrectable and failing sector count, total data read and written, and others. SMART is also capable of performing a variety of online and offline tests to verify the most visible problems related to drive operation.

Due to the nature of their implementation and their broad range of performance characteristics, HDDs are described using a number of parameters that help determine their usefulness for a specific application (Table 17.1). Many of these metrics also apply to other storage devices.

- *Storage capacity* is typically expressed in gigabytes or terabytes. Unlike memory capacities it is measured in powers of 10, hence 1 TB is 10^{12} bytes. HDD manufacturers tend to round this figure up. Note that due to storage of file system metadata, the data capacity available to users is 1%—5% less than the nominal capacity of the drive.
- *Seek time* (in milliseconds) expresses the duration taken by the read—write head to move to a specific cylinder. Average seek time is determined statistically as travel distances over one-third of all tracks on a disk. Of specific interest are also track-to-track latency (moving the head between adjacent tracks) and full-stroke latency, which involves travel between the innermost and outermost cylinders. They describe respectively the shortest and longest possible seek times.
- *RPM* is the number of rotations the platters perform in 1 min.
- *Rotational latency* (in milliseconds) describes the time required to position a specific sector under the read—write head. Average latency is typically given as the time it takes the drive to perform half a rotation of the platter, and is directly dependent on its RPM rating.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 505

Table 17.1 Comparison of Characteristic Hard Disk Drive Properties From Several Manufacturers

Manufacturer and Drive	Capacity (TB)	Media Transfer Rate (MB/s)	Seek Time (ms)		RPM	Cache (DRAM/ flash) (MB)	MTBF (million hours)	Average Power (W)		UER	Acoustic Noise [dB(A)]		Form Factor (inches)	Market Segment
			Track to Track	Full Stroke				Seek	Idle		Seek	Idle		
WDC WD101KRYZ	10	249			7200	256/0	2.5	7.1	5.0	<1 in 10^{15}	36	20	3.5	Enterprise
WDC WD60EZRZ	6	175			5400	64/0		5.3	3.4	<1 in 10^{14}	28	25	3.5	Economy desktop
HGST HTS541010A9E680	1	124	1	20	5400	8/0		1.8	0.5		26	24	2.5	Mobile
Seagate ST10000VX0004	10	210				256/0	1	6.8	4.42	<1 in 10^{15}			3.5	A/V streaming
Seagate ST2000DX002	2	156	<9.5 (average)			64/8192		6.7	4.5	<1 in 10^{14}			3.5	Performance desktop

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 506

- *Access time* (in milliseconds) is the delay between the time a request for data is submitted by the host and the time data is returned by the drive. It is a compound metric involving a combination of rotational latency and seek time, typically determined through a synthetic benchmark that exercises various access scenarios.
- *Media transfer rate* (in megabytes per second) measures how fast the signal processing chain and controller can read the data from the platter.
- *Burst rate* (in megabytes per second) describes how fast the data may be transmitted between the host and the disk cache using transfers that do not exceed the cache capacity.
- *Areal density* (in gigabits per square inch) provides the achievable upper limit of information density per surface area on a recordable medium. Related metrics involving linear densities are tracks per inch and bits per inch.
- *Mean time between failures* (MTBF, in millions of hours) estimates a drive's resilience to nonrecoverable faults.
- *Uncorrectable error rate* (UER, no unit) estimates the probability of receiving data containing a hard error, i.e., an error that could not be either detected or fixed by the built-in ECC mechanisms or could not be corrected through operation retries.
- *Power consumption* (in watts) describes average energy requirements of a drive in several possible scenarios: during regular operation, during power-up (spin-up), while idle, and during standby. The latter may involve several levels of inactivity (sleep modes) that are particularly relevant to mobile and other battery-powered devices.
- *Acoustic noise* (in dB(A)) provides an upper bound on noise level produced by the device during active operation.
- *Shock resistance* (in g) describes a device's resilience to external mechanical impact. Typically two figures are given, for nonoperating and operating modes (the first is often orders of magnitude higher due to the robust protection mechanisms used in powered-off devices). The figures are often accompanied by test conditions specifying the shock duration or whether it was repeating.
- *Size* (in inches) provides mechanical dimensions of the drive so that proper enclosure may be adopted for its use.

17.3.2 SOLID-STATE DRIVE STORAGE

Advances in semiconductor technology enabled practical realization of high-capacity persistent storage in solid-state devices. The most broadly utilized SSDs today are the descendants of electrically erasable programmable read-only memory (EEPROM) technology, introduced by Toshiba in 1984. EEPROMs can store small amount of data using floating-gate metal oxide semiconductor (FGMOS) arrays. The FGMOS transistors are similar to regular field-effect transistors with oxide isolators, but they sandwich additional electrodes between the oxide layers above the channel. During the programming cycle (Fig. 17.6A), sufficiently high potential applied to the control gate causes the transistor to conduct. Applying a relatively high source-drain voltage causes some high-energy channel electrons to overcome the oxide barrier and "jump" to the floating gate in a process called hot electron injection. After the removal of programming voltage, the charge remains trapped on the floating electrode, thus creating an additional electric field that may modulate the width of the transistor's channel. By putting suitable voltages on the control gate and drain (much lower than those needed for programming), the channel source-drain resistance reflects the amount of charge stored on the floating

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 507

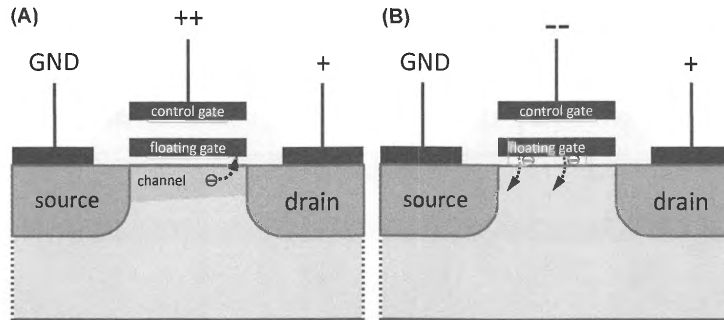


FIGURE 17.6

FGMOS transistor: (A) programming by hot electron injection and (B) erasing through quantum tunneling.

(B) From Smart Card Handbook, W. Rankl and W. Effing, third edition © 2002 Carl Hanser Verlag, München

gate. The erase process (Fig. 17.6B) requires negative voltage on the control gate and positive potential on the source and drain electrodes to cause Fowler—Nordheim tunneling of the trapped charge to the transistor body. Some variants of EEPROM used quantum tunneling for both programming and erasure. EEPROMs usually provide fine-grain access to storage, typically organizing data in groups equal in size to the width of a data bus (8 or 16 bits), but their capacities rarely exceed a few megabits. Early EEPROMs were frequently unable to support a fine-granularity erase function, instead supporting erasure of the entire chip or significant portions thereof. Future implementations alleviated this limit.

Increasing the device capacity necessitated reduction of the control structure and the number of internal connections, resulting in two dominant flash memory types: NOR and NAND. Their respective layouts are illustrated in Fig. 17.7. The names of flash configurations are derived from internal structures resembling that of NOR gate with a parallel arrangement of output n-type transistors and series connection of n-type transistors in the NAND gate. While the NOR configuration is nearly directly derived from the initial EEPROM structure, the NAND cell was proposed in 1987. Due to associated changes in storage transistor architecture (e.g., split gates and multiple-control gates) and their size, various flash operations became faster and more power efficient. This is particularly true for the erase operation, which could take as many as several seconds for EEPROMs but requires only a few tens of milliseconds for the NOR flash and single milliseconds for a NAND flash.

All flash memories are susceptible to several issues that negatively affect the reliability of general operation and data retention in the floating gate. The first is charge leakage, which may be caused by oxide (isolator) defects, electron detrapping, or contamination, in which positive ions present in the memory cell may in part offset the charge stored on the floating gate. The others are called *disturbs*, and may occur in neighboring cells that share some electrical connections with the cells that are programmed or erased (gate and drain disturbs). Moreover, as the electrical erase operation is not self-limiting, an extended erase cycle may leave a net-positive charge on the floating gate. This effect is

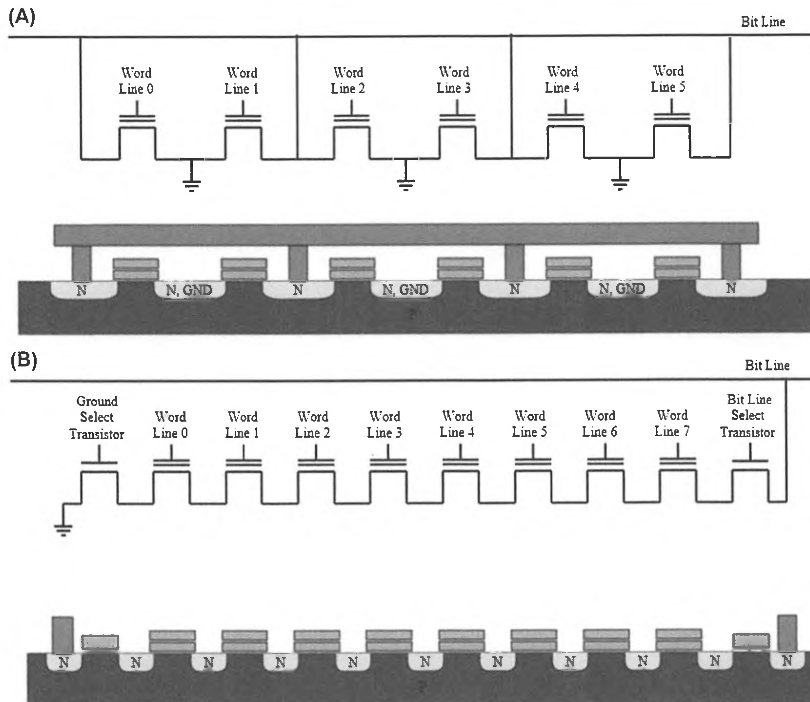


FIGURE 17.7

Storage cell connections and corresponding hardware implementation of (A) NOR flash memory and (B) NAND flash memory.

Both diagrams by Cyferz via Wikimedia Commons

called *overerasing*. The inverse phenomenon, *overprograming*, is also possible. The speed of various operations on flash storage is affected by its organization; the main features of NOR and NAND flash memories are compared in Table 17.2.

As can be seen, neither of the currently available flash technologies is ideal for mass storage. While the ability to manufacture high-capacity storage devices cheaply is paramount, selection of NAND memory is associated with serious drawbacks. The first is the relatively low number of update cycles that can be performed. Modern devices cope with this by application of wear-leveling algorithms that distribute the updates across all physical data blocks in a device by performing on-the-fly remapping of

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 509

Table 17.2 Comparison of Principal Properties of NOR and NAND Flash Memory

Property	NOR Flash	NAND Flash
Capacity	Low	High
Cost per bit	High	Low
Read speed	High	Medium
Write speed	Very slow	Slow
Erase speed	Very slow (10–100 s of ms)	Medium (single ms)
Erase cycles (endurance)	100,000–1,000,000	1000–10,000
Active power	High	Low
Standby power	Low	Medium
Random access	Easy	Hard
Block storage	Medium	Easy

logical addresses of rewritten blocks to physical addresses of the least utilized available blocks. This also means that many applications which take for granted multiple rewritability of stored data (such as when using HDDs) should not be used without caution. A good example is data wiping performed by repeated in-place overwriting of the file contents with pseudorandom data to prevent the reconstruction of its contents; due to wear leveling, it is completely ineffective in an SSD. The jury is still out on whether installation of a swap partition on a flash device is a good idea. It may substantially boost the performance compared to HDD-based solution; but while for relatively lightweight paging duties such an arrangement should not cause measurable problems, its use in severely memory-constrained systems may lead to premature failure of the flash drive.

Charge leakage is one of the main factors limiting the miniaturization of storage cells. Moreover, the amount of charge per cell cannot be decreased indefinitely. Thus in recent years the industry transitioned to encoding multiple bits per cell. Originally, NAND storage used single-level cell (SLC) implementation; commercial devices available today resort to multilevel cells (MLCs) with 2 bits per cell and even triple-level cells (TLCs) with 3 bits stored in a cell, thus representing eight data states. Sizes of cells used in MLC and TLC devices are somewhat larger than those of SLC to provide a reasonable margin of error despite leakage and disturbs. To ensure reliability, flash-based solid-state storage employs Bose–Chaudhuri–Hocquenghem codes [12] for error detection and correction. These enable correction of 24-bit errors in each 1024-bit sequence (two data sectors), with about 4% encoding overhead. Even then, endurance of TLC devices drops effectively to about 3000 erase cycles.

To operate correctly, SSDs require controller logic in addition to flash memory circuits. The controller interfaces to the host computer, typically using common high-bandwidth buses such as Serial AT Attachment, PCI Express (PCIe), or their variants (mini-PCIe, M.2, etc.). The controller must support a data buffer implemented in fast memory (DRAM) to cope with relatively slow performance of erase and write operations by individual chips. Since the storage pool is organized into multiple banks, the controller takes care of proper interleaving and overlap of the data accesses to extract maximum bandwidth from the pool. Mapping of logical to physical data blocks and wear leveling are also handled by the controller. Due to charge leakage, the read data has to be verified against ECC; if errors are detected, the corrected bit values are computed and written back to the

storage. The controller may periodically check for data that has been residing a long time in storage without being accessed to ensure it is viable; this function is called data *scrubbing*. Finally, the controller takes care of block allocation for new data, in many cases interpreting block usage hints from the OS, such as TRIM commands.

Table 17.3 presents examples of commercially available SSD devices with their parameters. Unlike HDDs, SSDs do not have moving parts, hence there is no equivalent of RPM or seek time. However, since solid-state storage handles multiple short accesses with much better performance, the number of I/O operations per second (IOPS) is given. The limited rewrite count of a flash is reflected through the terabytes written (TBW) statistic, which estimates the total volume of data a drive is guaranteed to accommodate over its lifetime taking into account wear leveling. Alternatively, some manufacturers may specify diskful writes per day over the warranty period of the drive. Independent tests verify that most SDDs significantly exceed this parameter in typical usage environments, with the possible exception of applications in which the drive is updated in low temperatures (significantly below room temperature) and stored in powered-off state for extended time at an elevated temperature (e.g., 50°C). Fig. 17.8 shows photographs of the devices listed in Table 17.3.

17.3.3 MAGNETIC TAPE

Magnetic tape has a long history as a computer data storage medium. Having been used as secondary storage (manufactured by Uniservo) in UNIVAC in 1951, tape predates HDDs by approximately 5 years. It consisted of 0.5″ wide and 0.0015″ think nickel-plated phosphor bronze metal tape wound on open reels, was up to 1500 ft long, and recorded information at the density of 128 bits per inch. The sustained data bandwidth was 7200 characters per second. A single reel with tape weighed about 3 pounds.

Later developments introduced polymer-based tapes, such as cellulose acetate, incorporating ferrous oxides as the magnetic recording medium. The IBM 726 shown in Fig. 17.9A is the iconic example of mid-1950s tape storage technology. The data was recorded in seven parallel tracks (including one-bit parity for ECC) on tape that could be read forwards and backwards. The tape could start advancing or reach a full stop in much less than 10 ms thanks to an innovative "vacuum column" arrangement that avoided the use of slower conventional tape-tensioning mechanisms. The maximum per-reel capacity was about 2 million six-bit characters.

Besides increases in data density and length of tape stored on a reel, improvements in tape and deck technology brought more practical implementations of replaceable storage media. Instead of using independent reels, they were packaged into *tape cartridges* that combined reels, tape, and some elements of a guiding mechanism into a single enclosure. An example is IBM's 3840 tape format (Fig. 17.9B) and its later derivatives. IBM's compatible tape storage was also manufactured by other vendors, such as Fujitsu, M4 Data, StorageTek, VDS, and Overland Data. But a lack of widely accepted standards for tape storage resulted in a proliferation of mutually incompatible cartridge families, including DDS (digital data storage, from 1989), DAT (digital audio tape, originated in 2003), DLT (digital linear tape, 1984–2007), and finally LTO (linear tape-open, 2000–today). Example cartridges and supporting tape decks are shown in Fig. 17.10. There were several iterations of capacity and resultant cartridge formats within each family; with some exceptions (e.g., DLT value line, or DLT-V), the newer releases are explicitly not backwards compatible with the products of earlier generations in each product line.

Table 17.3 Examples of Currently Manufactured SSD Devices and Their Operational Properties

Manufacturer and Device	Capacity (GB)	Sequential Read (MB/s)	Sequential Write (MB/s)	Maximum 4 KB Random Reads (kIOPS)	Maximum 4 KB Random Writes (kIOPS)	Terabytes Written	MTTF (million hours)	Power (Active/ Idle) (W)	Memory Type	Interface
Crucial CT2050MX300SSD1	2050	530	510	92	83	400	1.5	0.15 (average)	3D TLC NAND	SATA 6 Gbps
Samsung MZ-V6P2T0BW	2048	3500	2100	440	360	1,200		(5.8/1.2)	48-layer MLC V-NAND	NVMe 1.1, PCIe 3.0 × 4
SanDisk SDFADCMOS-6T40-SF1	6400	2800	2200	285	385	22,000		25 (peak)	MLC	PCIe 2 × 8

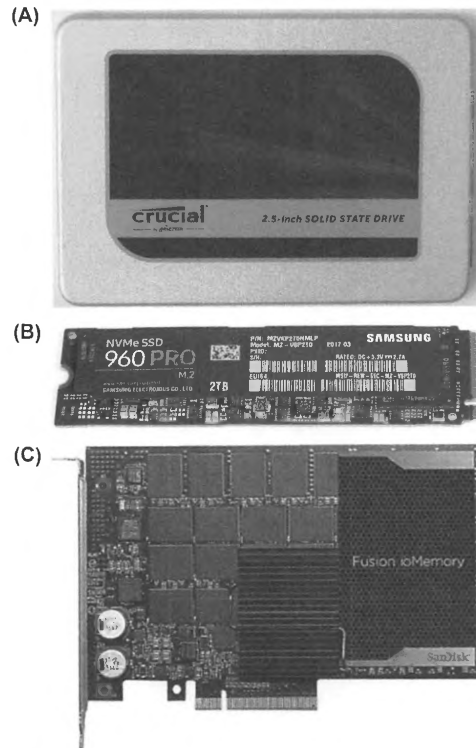


FIGURE 17.8

SSD examples: (A) Crucial MX300 series (2.5″ form factor), (B) Samsung 960 PRO series (M.2 form factor), and (C) SanDisk Fusion ioMemory SX350 series (8-lane PCI Express card).

(B) Photo by Dmitry Nosachev via Wikimedia Commons

The tape is a sequential-access medium, which means that it may require a comparatively lengthy time to locate a specific piece of data. The information on tape can be arranged in several ways. The earliest approaches used linear multitrack recoding, illustrated in Fig. 17.11A. In this mode, each read–write head records data lengthwise in a separate linear data track; the tracks are parallel to each other. As the bit density increased and track width decreased, linear-serpentine recording (Fig. 17.11B) permitted installation of several read–write heads side by side without loss of medium coverage. The head assembly moves across the width of the tape to start a new track in unrecorded space whenever

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 513

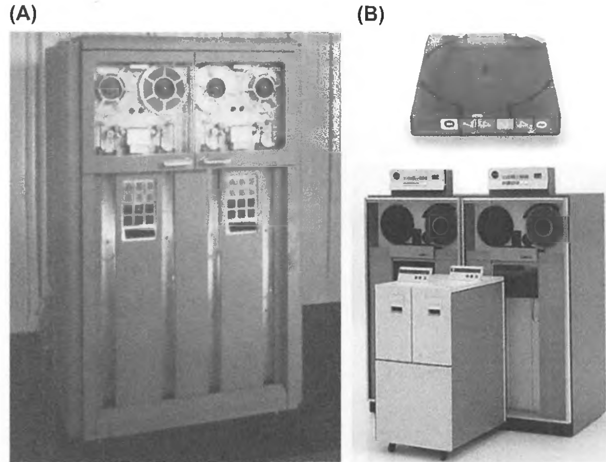


FIGURE 17.9

Advances in magnetic tape storage: (A) IBM 726 from 1951, (B) IBM 3480 format tape (top) and the corresponding deck subsystem (bottom) from 1984, with older 3480 system in the background.

(A) Courtesy of International Business Machines Corporation, © International Business Machines Corporation. (B) Bottom: Courtesy of International Business Machines Corporation, © International Business Machines Corporation

the tape reaches one of its ends. Helical recording, shown in Fig. 17.11C, arranges a large number of relatively short data tracks at an angle to the tape's edge. This last approach, resembling the recording technology used by tape-based camcorders and videocassette recorders, requires the use of a *scanning head* (a revolving drum which contains one or more heads along its circumference and is mounted at an angle to the tape's movement).

The longest-surviving technology still popular today is LTO, established in response to proprietary tape formats and developed by a consortium founded by Hewlett-Packard, IBM, and Quantum. Its current generation, LTO-7, supports up to 6 TB of data per cartridge packed on a 960 m long, 12.65 mm wide, and 5.6 μm thick tape. The tape substrate is polyester-based (polyethylene naphthalate), encasing particles of barium ferrite pigment as the active storage medium. The data is recorded in a linear-serpentine fashion on four data bands interleaved with five narrow servo (positioning) bands. Each data band is further subdivided into 28 wraps. There are 32 tracks per wrap (the same as the count of read—write elements in a head assembly), hence the total number of tracks stored on a tape is $4 \times 28 \times 32 = 3584$. The number of head passes required to fill the tape completely is a product of the data band count and the wrap count or 112; a data band is usually completely filled before the mechanism advances to the next band.

Table 17.4 compares operational parameters (only uncompressed data rates and capacities are reported) of some currently available tape decks. Their primary applications are backups and archivization of large datasets.

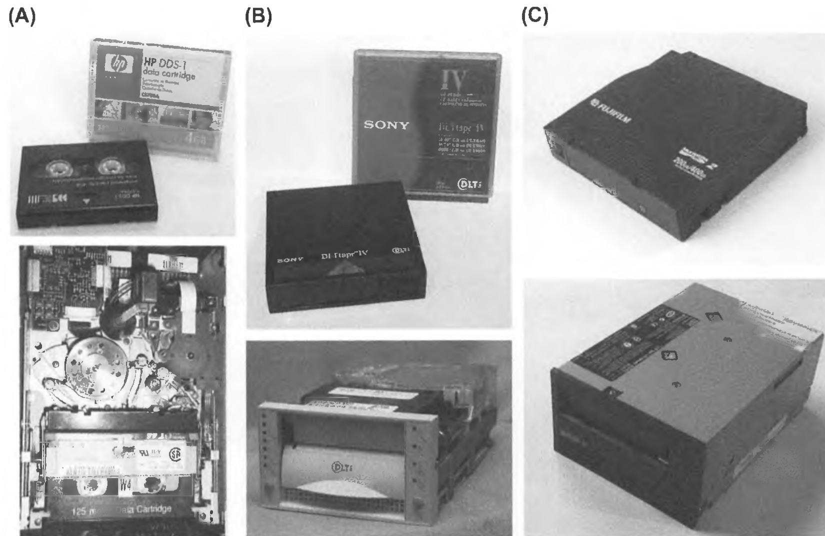


FIGURE 17.10

Comparison of dominant tape storage families: (A) DDS-1 (1989), (B) DLT-IV (1999), and (C) LTO-2 (2005). Data cartridges are shown on top and the corresponding tape decks at the bottom in each column.

(A) bottom photo by Adlerweb via Wikimedia Commons; (B) bottom photo by Christian Taube Chtaube via Wikimedia Commons; (C) Bottom and top figures by Austin Murphy via Wikimedia Commons

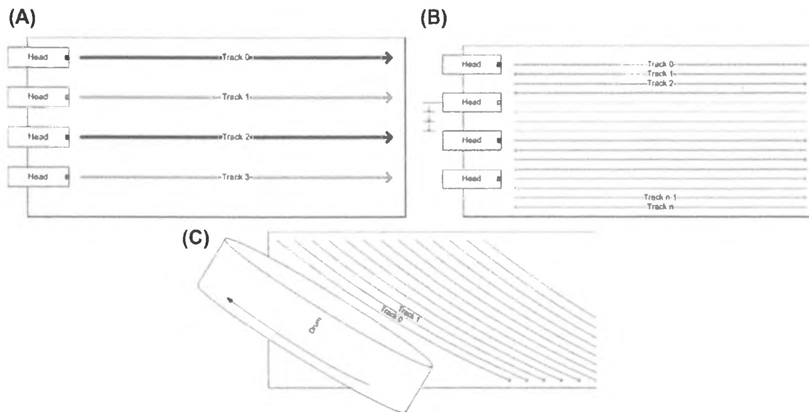


FIGURE 17.11

Tape-recording formats: (A) linear, (B) linear-serpentine, and (C) helical.

Diagrams by Kubanczyk at the English language Wikipedia

Table 17.4 Operational Parameters of Selected Tape Drives

Manufacturer and Drive	Capacity (TB)	Sustained Data Rate (MB/s)	High-Speed Search (m/s)	Maximum Operating Power (W)	Data Format	Cartridge Types Supported	Interface
IBM TS1150	Up to 10 (medium dependent)	360, 300	12.4	46	32-channel linear-serpentine	IBM 3592 Generations 3 and 4	8 Gbps fiber channel
HP Enterprise BB873A	Up to 6	300			32-channel linear-serpentine	LTO-7 (rewritable), LTO-6 (rewritable), LTO-5 (read only)	6 Gbps Serial Attached SCSI

17.3.4 OPTICAL STORAGE

While there were many attempts to apply optical means for storage and retrieval of digital information, none attained widespread popularity before the commercial release of the compact disc (CD) in 1982. The CD is the result of a collaboration between Philips and Sony, who jointly developed the Red Book CD digital/audio specifications and agreed to manufacture compatible hardware. Even though originally intended as a medium for music distribution, the CD was soon used to store photographs, graphics, artwork, sound samples, video, and, of course, data. As early versions did not support recording data, the information stored on the disks was read-only and inspired the CD-ROM moniker describing media carrying digital data. Starting in the 1990s and continuing to this day, CD-ROM and its derivatives are extensively used as an inexpensive medium to distribute software and other auxiliary data.

Physically, a CD is a 1.2 mm thick plastic disc with a diameter of 120 mm. The base material is polycarbonate with an impressed spiral pattern of elongated pits to encode the data. The data track is covered with a reflective metal layer (usually aluminum or occasionally gold) before sealing it with a protective layer of lacquer and artwork (Fig. 17.12B). The information is retrieved from the spinning disc using an infrared laser beam equipped with appropriate collimating optics and tracking mechanism (Fig. 17.12C). Most discs have just one active surface that is used to read the data, although there are variants with information recorded on both sides. A smaller 80 mm version of the disc called a mini-CD is also in circulation. While a conventional CD stores 74—80 min of audio or up to 700 MB of data, a mini-CD reduces this to up to 24 min of music and approximately 200 MB of data. On audio disc, two channels of sound are sampled at 44.1 kHz using linear 16-bit encoding (the complement integer of two) per sample per channel. The audio data is organized in 192-bit *frames*; each frame contains six interleaved audio samples from left and right channels. In addition to audio samples, frames incorporate ECC and synchronization data. Due to symbol transcoding to reduce the density of pits on the disc surface (eight-to-fourteen modulation code [13]), each frame effectively ends up occupying 588 bits on the disc. Frames are combined into *sectors*, each containing 98 frames or 2352 bytes of audio data. The sectors are assigned to *tracks* that correspond to individual songs on the

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 516

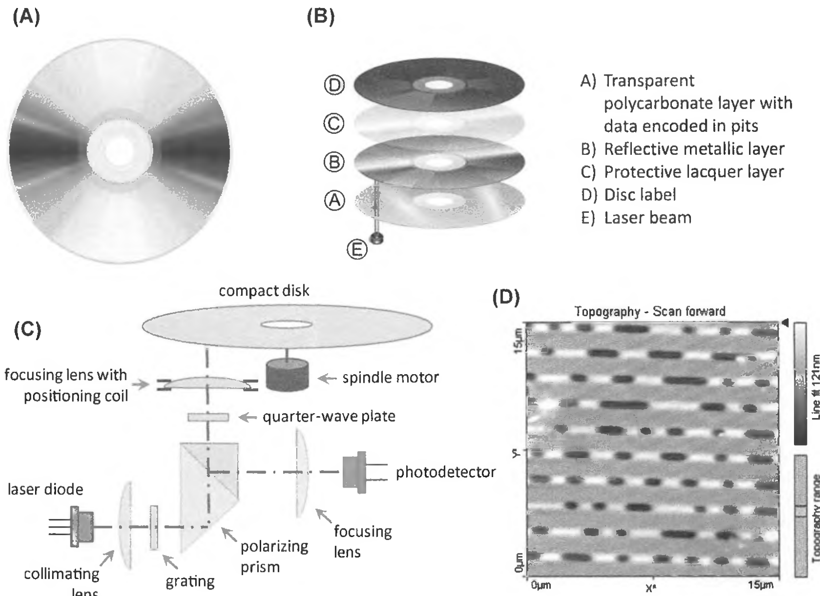


FIGURE 17.12

Compact disc: (A) medium, (B) component layers, (C) optical pickup mechanism, and (D) geometric properties of the data track.

(A) Image by Arun Kulshreshtha via Wikimedia Commons; (B) Image by Pbroks13 via Wikimedia Commons; (D) Image by Valacosa and Blair Lebert via Wikimedia Commons

CD; up to 99 tracks may be stored on one disc. The nominal data rate is 2 (channels) × 2 (bytes per sample) × 44,100 (samples per second) = 176.4 kB/s; this is equivalent to a throughput of 75 sectors per second. Data integrity is protected by cross-interleaved Reed–Solomon code (CIRC) [14], which adds one parity byte for every three bytes of data. CIRC is capable of correcting up to two full byte errors in each 32-byte block, or, due to interleaving of parity data with the neighboring blocks, it can fully correct up to 4000-bit-long error bursts corresponding to 2.5 mm in linear distance. This makes it a very effective solution to deal with scratches, particulate matter, and small stains on the disc surface.

For data storage, a CD-ROM retains the basic organization of information on the disc, but the effective number of data bytes per sector is reduced to 2048 (CD-ROM Mode 1) due to the stronger ECC schemes employed (audio data may be reconstructed to some extent by interpolation, but this is

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 517

not true for arbitrary digital information). For some applications, like video, robust protection is less important than data density, hence CD-ROM Mode 2 specification permits 2336 data bytes per sector. The base data rate, referred to as 1× speed, is derived as the data throughput of 75 CD-ROM Mode 1 sectors per second or 153.6 kB/s. Many currently manufactured CD-ROM drives are capable of spinning the disc much faster than that, resulting in sustained transfer rates of 24×–48× and higher.

One of the main drawbacks of a CD-ROM is that its content is fixed at the factory, essentially precluding its use as a practical mass-storage medium. This has been addressed by CD-recordable (CD-R) and later CD-rewritable (CD-RW) formats, detailed by Orange Book specifications. Both retain the original form factor of the 120 mm disc. CD-R media replace data-defining pits with a fixed spiral "pregroove" to aid laser positioning and add a layer of organic dye between the polycarbonate substrate and reflective layer. During the write process, the laser power is modulated to affect ("burn") the organic dye, making it locally more opaque or absorptive. The read is performed at much lower beam power so the written data is not destroyed. As the mass-produced media adopt primarily three (cyanine, phtalocyanine, and azo) dyes of quite different properties, careful calibration of laser power is required ahead of data deposition. This process is aided by the additional information (absolute time in pregroove) stored on the blank disc in the pregroove outside the useful data area, which identifies media manufacturer as well as the recommended laser power. CD-R discs may be "burned" only once, but depending on the write mode selected it may be possible to add data at a later time to a disc that has not been "closed" (*track at once* mode as opposed to *disk at once* mode). Since some of the dyes are sensitive to ultraviolet light, ensuring proper storage conditions is strongly encouraged to achieve the desired information longevity. Good-quality CD-R media recorded in properly calibrated devices and stored in a dark location with stable temperature and humidity may last over 50 years without data loss; archival-quality discs using gold as the reflective layer may extend this to as much as 100 years. A National Institute of Standards and Technology study estimated the longevity of several tested media brands to be at least 30 years if kept at ambient temperature and controlled humidity conditions [15]. Rewritable discs utilize silver-indium-antimony-tellurium phase-change media that may alternate between crystalline and amorphous phases differing in reflectivity. Thus the composition of CD-RW discs is similar to that of regular CD-ROMs, but with a different material constituting the reflective layer. As the nominal reflectivity of phase-change media is much lower than that of aluminum or gold, the recorded CD-RW media may not always work correctly in older CD-ROM drives. CD-RW discs require even more precise control of laser power while writing than CD-R, and constrain both the upper and lower limits of data transfer rate while burning. Rewritable media endurance is commonly estimated at approximately 1000 rewrite cycles. Since CD-RW media can be updated and erased, *packet writing* mode has been developed to support changes to the stored information.

The maximum data capacity of a CD is insufficient to store a full-length movie in National Television System Committee resolution, even using a lossy compression such as MPEG-2. To cope with the growing demand for multimedia content and simultaneously increase the storage capacity of disc-based media, Philips, Sony, Toshiba, and Panasonic introduced the DVD (digital versatile disc, alternatively known as digital video disc) in 1995. While DVDs have the same external dimensions as CDs, the information is retrieved using a red laser of wavelength 650 nm, which permits reducing the gap between the consecutive windings of data "groove," thus making it much longer. DVDs may store data in one or two layers; this results in a total capacity of 4.7 GB or 8.5 GB per disc. The nominal (1×) data rate is 1385 kB/s; note that this reference speed for DVDs is substantially higher than that of CDs. Modern DVD-ROM drives may retrieve the data at a rate that is 8–20× greater than the base rate.

Similar to CDs, DVDs support recordable and rewritable variants. Due to "format wars" there are two recordable versions, DVD-R and DVD + R and two rewritable versions, DVD-RW and DVD + RW. The "-" and "+" formats are not directly compatible with each other, but most drive manufacturers release products that support both. Since the -R and -RW formats originally developed by Pioneer were released earlier, they are supported by more devices, especially standalone DVD video players. The "+" variants specified by Sony and Philips feature more robust error correction, hence they may be somewhat more suitable for data preservation. Additionally, the DVD-RAM (digital versatile disc-random access memory) format backed by Hitachi, Toshiba, Maxell, Samsung, LG, Panasonic, Lite-On, and Teac offers very good support for data updates (minimum of 100,000 rewrites at low speeds), protection, and retention. Unlike other recordable DVD discs, the DVD-RAM stores data in concentric tracks, similarly to HDDs, and therefore requires specialized drives.

Widespread adoption of high-definition (HD) video formats prompted the development of a suitable storage medium. Of two competing variants, HD-DVD and Blu-ray disc (BD), the latter ultimately emerged as a winner in 2008. Blu-ray media can store 25 GB per layer thanks to the availability of violet laser diodes operating at 405 nm wavelength. This permitted shrinking the track pitch further from 740 nm for DVDs to just 320 nm (Fig. 17.13). BD differs from other optical disc technologies in that the data tracks are much closer to the surface and thus are more vulnerable to scratches. A specially formulated hard-coat layer applied to the top surface alleviates the effects of mechanical impact. The 1× speed for Blu-ray is equivalent to real-time reproduction bandwidth for compressed 1080p video at 60 frames per second, and equals 4.5 MB/s. Practical speeds achieved by currently manufactured drives range from 4× to 16×. Data capacities per disc range from 25 to 50 GB for single-layer and dual-layer media to 100 GB (triple-layer) and 128 GB (quad-layer) BDXL discs. Example optical drive specifications are listed in Table 17.5.

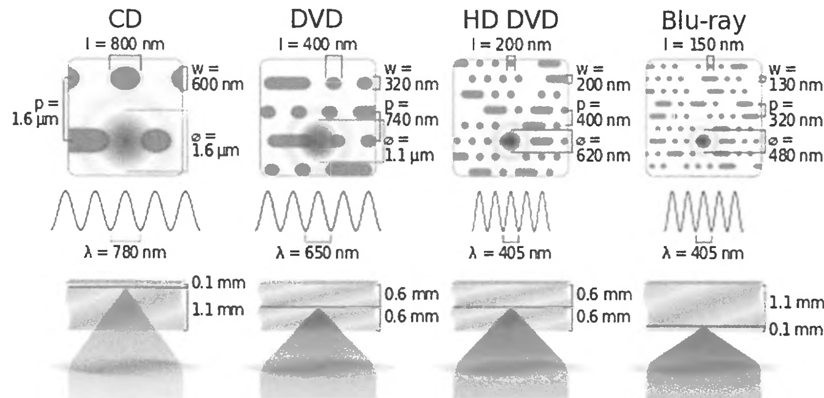


FIGURE 17.13

Comparison of optical format geometries (CD, DVD, HD-DVD, and Blu-ray). The listed parameters denote minimum feature length (l), track width (w), track pitch (p), laser beam diameter (φ), and wavelength (λ).

Diagram by Cmglee via Wikimedia Commons

Table 17.5 Parameters of a Typical Consumer-Grade Multiformat Optical Drive

Manufacturer and Drive	BD access Time (ms)	DVD access Time (ms)	CD access Time (ms)	Maximum Data Rate						Buffer Size (MB)	Interface
				BD Read	BD Write	DVD Read	DVD Write	CD Read	CD Write		
Lite-On iHBS312	250 (SL) 380 (DL)	150 (ROM) 160 (DL) 200 (RAM)	150	6× (RE DL) 8× (SL)	2× (rewrite) 8× (DL) 12× (SL)	16×	6× (rewrite) 12× (RAM) 16× (+R, −R)	48×	24× (−RW) 48× (−R)	8	SATA (internal)

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 520

17.4 AGGREGATED STORAGE

17.4.1 REDUNDANT ARRAY OF INDEPENDENT DISKS

Redundant array of independent disks (RAID; formerly redundant array of inexpensive disks, attributed to David Patterson, Garth Gibson, and Randy Katz of the University of California at Berkeley) attempts to address reliability issues of conventional mass-storage devices. All storage devices, including HDDs and SSDs, have a limited lifespan and undergo random mechanical or electrical failures. The consequence of a failure is usually loss of a portion or the whole amount of the data stored on the device. RAID works by extending the pool of drives containing actual data with additional devices. These *redundant* drives store information that is derived from the contents of other drives in the pool. By treating such a formed array of drives as a single, virtualized I/O device, the impact of individual component failures may be alleviated. However, RAID should never be considered a perfect or universal solution. It may mitigate component failures only to a certain extent, which is defined by RAID level, implementation, parameters of component drives, and even their fabrication characteristics. Since drives in an array are accessed in aggregate, in many cases RAID usage translates into improved data access performance compared to that of a single device. Here a number of commonly recognized RAID configurations are discussed, along with their main operational properties.

17.4.1.1 RAID 0: Striping

RAID 0 is not a proper RAID level, in that it does not provide any data redundancy should drive failures occur. It describes a configuration in which the data blocks are simply distributed (striped) across available disks in a round-robin fashion, as shown in Fig. 17.14. A *stripe* is a sequence of blocks spanning all disks in the array; for example, block 4—block 5—block 6 in the figure constitutes a stripe. An arbitrary number of disks may be arranged this way, but assuming that failure occurrences are independent and have exponential probability distribution, the reliability for the whole array including d data disks will be a fraction of that for a single drive:

$$MTBF_0 = \frac{MTBF_D}{d}$$

Thus building an array of four enterprise drives, each with a good MTBF rating of 1,200,000 h, will result in an MTBF for the array of 300,000 h—equivalent to an average consumer drive. With

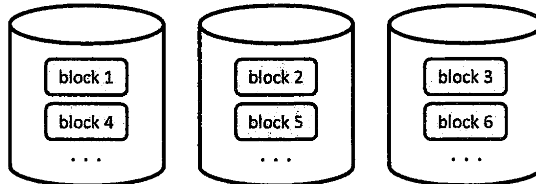


FIGURE 17.14

RAID 0 data layout.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 521

independent controllers, data on the drive may be accessed concurrently, providing increased read and write bandwidths in proportion to the number of drives:

$$B_{R_0} = d \cdot B_{R_D}$$
$$B_{W_0} = d \cdot B_{W_D}$$

where B_{R_D} and B_{W_D} are respectively read and write bandwidths of a single drive.

Finally, the storage capacity of the whole array is a sum of the component drive capacities:

$$C_0 = d \cdot C_D$$

where C_D is the capacity of a single drive.

17.4.1.2 RAID 1: Mirroring

RAID 1 is the lowest RAID level supporting data protection (Fig. 17.15). This is accomplished by storing replicas of used data blocks that reside on the primary data drive on all other drives in the array (data mirroring). While there is no upper limit on the number of drives arranged in this fashion, typical installations use just one redundant drive (mirror) in addition to the primary drive. Hence the number of data disks is fixed at $d = 1$; assuming a general case with p mirror drives, a RAID 1 array can tolerate up to p concurrent drive failures without data loss. It is worthy of note that read accesses can take advantage of all I/O devices to issue concurrent requests, thus effectively matching the throughput of RAID 0 with the equivalent number of devices. Write operations, however, need to store data replicas on all mirror drives in addition to the "regular" data drive, effectively achieving the write throughput equivalent of a single drive. The resulting formulae are:

$$d = 1, p \geq 1$$
$$B_{R_1} = (p + 1) \cdot B_{R_D}$$
$$B_{W_1} = B_{W_D}$$
$$C_1 = C_D$$

Due to its simplicity, mirroring is frequently used by both hardware and software RAID implementations, but its biggest drawback is 50% (or higher) storage overhead.

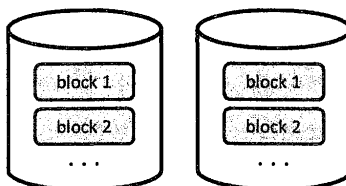


FIGURE 17.15

RAID 1 data layout.

17.4.1.3 RAID 2: Bit-Level Striping With Hamming Code

RAID 2 attempted to reduce spatial overheads caused by data mirroring by selecting a more efficient data protection code. Hamming code uses $p \geq 2$ code bits to protect each group of $d = 2^p - p - 1$ data bits against single bit errors, hence attaining $(2^p - p - 1)/(2^p - 1)$ efficiency or *code rate*. A RAID 2 array consists of d data drives and p parity (protection bits are calculated as parity for selected bits in the entire bit-stripe) drives. The minimum configuration consists of two parity drives and one data drive, although it has poor storage efficiency of 1/3; the efficiency vastly improves for larger ensembles. The drives have synchronized spindles, ensuring lock-step update and retrieval of each bit in individual stripes (denoted as a and b in Fig. 17.16). This arrangement is able to recover from a single device failure. Since hamming code can pinpoint the position of the erroneous bit in each stripe, RAID 2 is capable of detecting silent drive malfunctions in which the affected device may appear to work but returns invalid data. This property may also be used to correct occasional data errors on the fly due to the nonzero probability of uncorrectable read errors returned by individual disks. Due to the implementation complexity requiring specialized hardware controllers, RAID 2 is no longer used in practice. Its performance characteristic strongly depends on the implementation details, and hence is not analyzed here.

$$d = 2^p - p - 1, \quad p \geq 2$$
$$C_2 = d \cdot C_D$$

17.4.1.4 RAID 3: Byte-Level Striping With Dedicated Parity

RAID 3 further decreases the required number of redundant drives in the assembly. Unlike RAID 2, it performs byte-level striping. Just one extra drive ($p = 1$) is used to store the parity value for all bytes in the same stripe (Fig. 17.17). Since the parity alone cannot be used to identify a broken drive, RAID 3 recovery is activated after one of the devices overtly fails. In that case, the missing information (assuming the failed drive was not the parity drive) is reconstructed from parity information and the

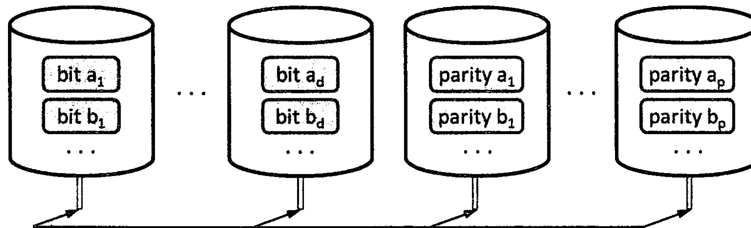


FIGURE 17.16

RAID 2 data layout.

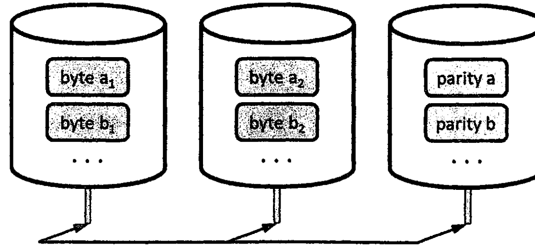


FIGURE 17.17

RAID 3 data layout.

remaining byte values in the corresponding stripe. Operation at byte granularity forced synchronized disk spindle control, similar to RAID 2. As RAID 3 does not offer specific advantages over higher RAID levels and requires specialized hardware support to work, it too was phased out from common usage. RAID 3 achieves good large-volume sequential read and write performance (see below), but lags for small or multiple simultaneous requests.

$$d \geq 2, \quad p = 1$$
$$B_{R_3} = d \cdot B_{R_D}$$
$$B_{W_3} = d \cdot B_{W_D}$$
$$C_3 = d \cdot C_D$$

17.4.1.5 RAID 4: Block-Level Striping With Dedicated Parity

RAID 4 (Fig. 17.18) eschews the fine-granularity synchronization of RAID 3, instead performing block-level striping across all data devices much like RAID 0. For recovery, one parity drive is used; its function is similar to the parity drive in RAID 3 except that parity information is computed on a per block basis. The minimum configuration consists of three devices (two data drives, one parity drive). The large request performance is good, since it can be satisfied with sequential access to multiple blocks on each data drive. Since the drives do not have to be synchronized, simultaneous small requests may be distributed over multiple devices, yielding improved IOPS figures.

$$d \geq 2, \quad p = 1$$
$$B_{R_4} = d \cdot B_{R_D}$$
$$B_{W_4} = d \cdot B_{W_D}$$
$$C_4 = d \cdot C_D$$

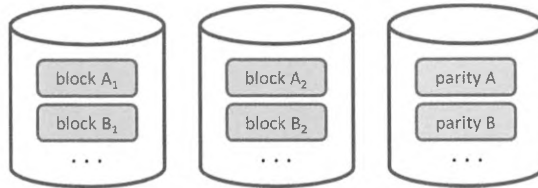


FIGURE 17.18

RAID 4 data layout.

17.4.1.6 RAID 5: Block-Level Striping With Single Distributed Parity

RAID 5 is one of the most commonly employed data protection schemes (Fig. 17.19). It shares many similarities with RAID 4 in terms of parity computation, access granularity, minimum configuration, and vulnerability to failures. The main difference is that there is no dedicated parity drive: the parity blocks are distributed in round-robin fashion across all participating devices. This modification enables the system to achieve high read bandwidths, effectively matching those of RAID 0 with an equal number of disks. The main issue of RAID 5 storage is its high vulnerability in a degraded state (i.e., after it has suffered drive failure). Even if the replacement drive is quickly furnished, the rebuild process for the whole array may take several hours. During that time the component drives are accessed at close to full bandwidth, exposing the remaining devices to increased stress levels. A second device malfunction during that time may effectively destroy the nonrebuilt fraction of data stored in the array.

$$d \geq 2, \quad p = 1$$
$$B_{R_5} = (d+1) \cdot B_{R_D}$$
$$B_{W_5} = d \cdot B_{W_D}$$
$$C_5 = d \cdot C_D$$

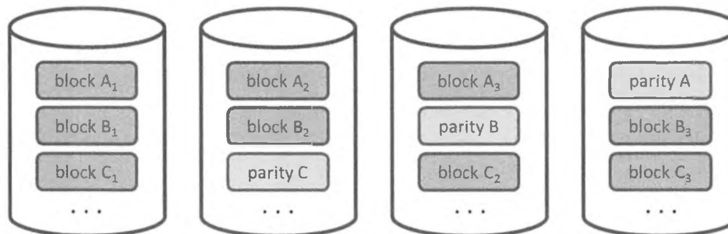


FIGURE 17.19

RAID 5 data layout.

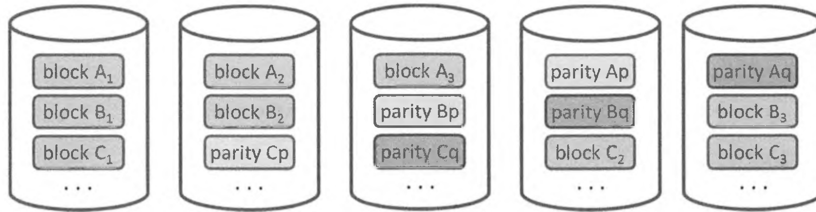


FIGURE 17.20

RAID 6 data layout.

17.4.1.7 RAID 6: Block-Level Striping With Dual Distributed Parity

To maintain array operation in a degraded state with no more than two failed drives, RAID 6 associates two parity drives with each group of data drives. Much like RAID 5, the parity blocks are distributed over all drives in the array. Parity information, denoted in Fig. 17.20 by indexes p and q, must be computed using different methods, e.g., conventional bitwise XOR on the original contents of the stripe and XOR on the stripe contents transformed by an irreducible binary polynomial selected using the Galois field [16] theory. Following a single drive failure, the array may be reconstructed using conventional parity, which is fast to compute. A double fault requires usage of both parity blocks per stripe or, if the simple parity block was stored on the faulty drive, the missing data may be recomputed from the available data blocks and the secondary parity information. The calculation of secondary parity is more involved, and may benefit from hardware implementation.

$$d \geq 2, \quad p = 2$$

$$B_{R_6} = (d+2) \cdot B_{R_D}$$

$$B_{W_6} = d \cdot B_{W_D}$$

$$C_6 = d \cdot C_D$$

17.4.1.8 Hybrid RAID Variants

The most common hybrid RAID configurations are illustrated in Fig. 17.21. RAID 10, also denoted RAID 1+0, combines data mirroring at a lower level and striping at a higher level. This provides the benefits of a simple-to-implement redundancy scheme (mirroring) with improved data access performance due to striping. The main drawback is the low storage utilization of 50%. The configuration shown in Fig. 17.21 can tolerate two drive failures (one per mirror group). This version of RAID is commonly implemented in hardware controllers, including low-cost solutions embedded in motherboard firmware. A variant of RAID 10 that improves storage utilization replaces the mirroring with RAID 5 at the lowest level, and is known as RAID 50. As the smallest number of devices supported by RAID 5 is three, the minimum layout of RAID 50 consists of six devices. In comparison to conventional RAID 5 with six devices, striping improves the write throughput, while the ability to tolerate one fault per redundancy group makes this arrangement substantially more resilient. Of course, either

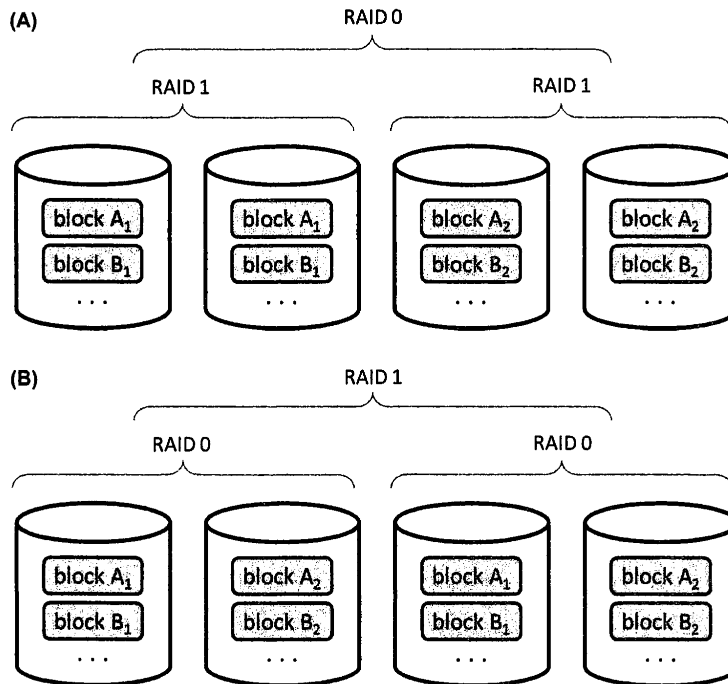


FIGURE 17.21

Diagram of data distribution: (A) RAID 10 (stripe of mirrors) and (B) RAID 01 (mirror of stripes).

RAID x0 configuration may include more than two components per stripe for further bandwidth boost, albeit at the cost of additional drives.

RAID 01 (or 0+1) has the same storage utilization, capacity, and access performance as RAID 10 with an equivalent number of mirrors and stripe units. However, while RAID 10 is still able to operate with one failed drive per mirror group, in RAID 01 loss of single drive equals the loss of the entire stripe. This has dramatic consequences for rebuild performance: RAID 10 can accomplish this by simply copying the contents of the remaining drive in its mirror group without disturbing other components of the array, while RAID 01 must pull the data from another functional mirror, interfering with its regular operation. RAID 01 has, however, practical applications when portions of the array are distributed over a network. Having a fully functional local RAID 0 setup is more important in the event of network outage than a mirror containing partial data.

Since the timely completion of array rebuild is often critical to stored data integrity, many RAID implementations include hot spares: idle drives that are connected to the controller, but do not actively share any part of the data. When a disk failure occurs, the controller may automatically switch to the replacement drive and start repopulating the array without having to wait for the system administrator. The failed drive may be pulled and replaced later at the operator's convenience.

Selection of component drives for the array has to be performed with special care. Good practice calls for verification that devices come from different fabrication batches to minimize the probability of correlated failures. Since malfunctions may also be related to other connected devices, avoiding sharing of critical components, such as power supplies, may prevent some failure modes. RAID-compatible drives typically support time-limited error recovery, which constrains the time spent by the drive on bad-sector recovery to prevent it from being marked by the RAID controller as unresponsive or faulty.

With high performance multicore processors being a common component of a node, many RAID modes do not require a specialized hardware controller to achieve good performance. Operating systems frequently offer optimized support for common levels (such as RAID 0, 1, 5, 6, and their hybrids) and sometimes nonstandard levels that nevertheless may provide well-performing redundant storage with less common drive counts and configurations. Software implementations may expose more configuration parameters, thus I/O benchmarking with different options is crucial to extracting maximum performance. They may, however, suffer from problems that are avoided by correctly designed hardware controllers; one such issue is the "write hole" caused by a system crash (e.g., due to power blackout) leaving parity information in an inconsistent state with the data on drives. Some file systems, such as ZFS developed by Sun Microsystems, support RAID-like data striping and protection without being vulnerable to this issue. Harnessing OS support to manage data integrity has additional benefits. Neither hardware controllers nor low-level software implementations are aware which portions of disk store the actual information, so upon failure the recovery algorithms must perform verification of data consistency on the entire drive, or at least the impaired partition. The same process guided by a file system's internal data structures may be far more efficient and focus only on the relevant areas of the disk. Prioritization is also possible, so the most critical file system metadata is recovered first. Since array rebuild places the system in a particularly vulnerable state, minimizing its duration additionally lowers the chances of unrecoverable failures.

17.4.2 STORAGE AREA NETWORKS

A storage area networks (SAN) provides a block-level storage abstraction over common networks (Fig. 17.22). Its purpose is to enable access to shared storage devices for multiple entities, including virtualized server pools or other hosts (e.g., related to management and monitoring infrastructure) attached to a common network. The shared storage devices may include HDDs or SSDs, optical jukeboxes, and tape silos. Clients connected to a correctly implemented SAN have an illusion of directly communicating with the attached storage devices, extracting close to the full available device bandwidth.

SANs offer many benefits to system administrators. Physical separation of servers and storage enables fast and independent replacement of failed components. Scaling in the number of storage devices as well as servers is usually easily accomplished. Application servers may directly boot from the attached drives, which minimizes the configuration time for newly added or replaced servers. Since

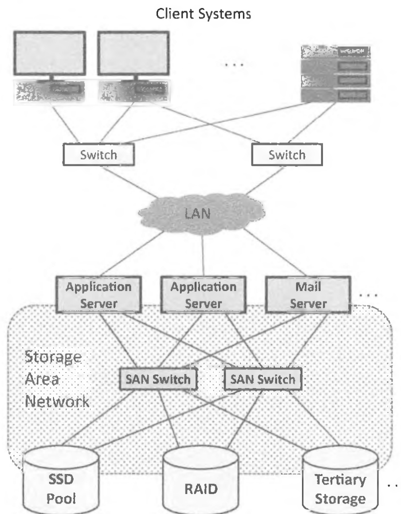


FIGURE 17.22

Example configuration of a storage area network.

the connecting network can span a large distance, significantly exceeding the length of interface link attaching individual storage devices, SANs are key to efficient disaster recovery. Storage contents can be replicated to different physical locations, with fast synchronization mechanisms already in place should major faults occur. SANs are frequently configured with multiple switches and redundant paths for increased reliability.

A SAN encapsulates lower-level access protocols to storage devices, such as SCSI (small computer system interface), in higher-level network protocols, primarily Ethernet, InfiniBand, and Fibre Channel (FC). The latter frequently utilizes optical fiber for connectivity, with communication rates between 1 and 128 Gbps. FC supports multiple topologies, including point to point, arbitrated loop, and switched fabric. Even though it has a reputation for being expensive and difficult to manage, it is often the preferred choice for SAN implementation. One of its main advantages is the asynchronous protocol design that is able to handle large numbers of data packets in a heavily loaded network. Different implementations of SANs utilize protocols that best fit the class of deployed network and low-level interfaces used by mass-storage devices. The number of combinations is quite large, but the dominant variants include FCP (Fibre Channel Protocol that encapsulates SCSI packets over FC), FCoE (Fibre Channel over Ethernet), HyperSCSI (SCSI over Ethernet), iFCP (FCP over IP), iSCSI (SCSI over Transmission Control Protocol/Internet Protocol (TCP/IP)), iSER (iSCSI extensions for RDMA in the InfiniBand network), SRP (a simpler SCSI RDMA protocol for transmitting SCSI over InfiniBand), AoE (ATA over Ethernet), and FICON (Enterprise Systems Connection over FC, used by mainframe machines).

17.4.3 NETWORK ATTACHED STORAGE

Network attached storage (NAS) is a common component of supercomputing installations. It provides centralized shared storage capability, frequently with very large capacity, to multiple hosts, specifically including compute and login nodes. While SANs provide shared access to mass storage at the device level, NAS operates at file-system level. The accessing clients use specifically designed libraries or kernel extensions to import data volumes hosted by NAS servers. Remote data shares may be mounted on the client side to provide practically identical application programming interfaces to those exposed by local file systems, such as Portable Operating System Interface I/O. The contents of remote data shares may be then accessed using standard utilities and libraries that have been developed to support "regular" files, effectively making the fact that the communication with the server and the data transfer are performed over the network completely transparent to the application.

NAS implementations utilize a handful of network file system protocols. The commonly used ones include Server Message Block (SMB), originally developed by IBM and Microsoft, Common Internet File System (CIFS), which is a more feature-rich version of SMB, Apple Filing Protocol (AFP), a proprietary protocol used by Apple File Service, and Network File System (NFS), which originated at Sun Microsystems. While the first two are usually found in Microsoft DOS and Windows-based environments, AFP is restricted to Apple products and NFS is broadly employed in the Unix world, including Linux. NFS is an open standard defined in the Internet Engineering Task Force/Internet Society Request for Comments and has open-source implementations. SMB functionality is available on Unix-compatible platforms thanks to the open-source SMB/CIFS reimplementation known as Samba. Finally, AFP is supported by the open source Netatalk project. All these protocols rely on TCP/IP for connectivity, although some SMB and NFS variants are capable of datagram-based communication (User Datagram Protocol).

A high performance NAS server, depicted in Fig. 17.23, derives from the architecture of a conventional compute node. The primary differences are possible inclusion of multiple network adapters to provide the necessary data bandwidth to clients and a substantially expanded storage pool. The latter usually requires multiple controller boards to provide the required number of ports for connecting the

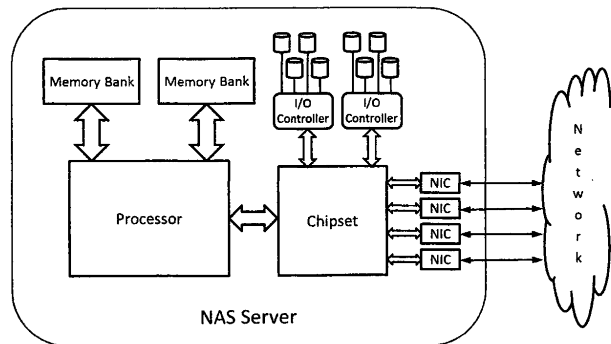


FIGURE 17.23

Simplified architecture of NAS server.

storage devices and optionally to incorporate hardware-level data protection, such as RAID. The server should have a sufficiently large memory pool to be able to accommodate a large number of outstanding I/O requests and efficiently buffer data. Due to the increased power draw caused by the large storage pool, a NAS server should also be equipped with redundant power supplies of appropriate rating and make allowance for sufficient case ventilation to evacuate the generated heat. Since a single server will eventually hit a performance barrier, a clustered NAS may be considered to enable capacity scaling. A clustered NAS takes advantage of distributed (Ceph, AFS, GFS, and others) or parallel (GPFS, Lustre, PANFS, OrangeFS, PVFS, and others) file systems to provide an abstraction of a single logical file system comprising all storage devices while enabling high-bandwidth access to file data and load distribution across the component servers.

17.4.4 TERTIARY STORAGE

Tertiary storage comprises high-capacity data archives designed to incorporate vast numbers of removable media, such as tapes or optical discs. The removable media are normally not stored in suitable drives but held in specially arranged retention slots, shelves, or carousels in an offline state. A tertiary storage platform may be perceived as a specialized type of NAS that uses additional robotic mechanisms to transfer media between their long-term storage locations and available drives without human intervention. To fulfill a client access request, a separate database that maintains the catalogue of archive contents must be consulted. As the tape library or optical jukebox cannot handle a large number of concurrent requests (there is only a limited number of tape or optical drives which operate at nominal data rates per device), the archive contents are typically copied to a data cache, for example a regular NAS server. Clients may then access the data at high speeds and possibly in parallel. The retrieved content is retained in the cache for as long as it is needed or until it is retired as an effect of the application of relevant data retention policies. Tertiary storage also performs periodic (or other policy-managed) scans of stored media to detect signs of content decay and possibly activate recovery procedures. Examples of two high-capacity tertiary storage systems, a tape library and an optical jukebox, are shown in Fig. 17.24 and compared in Table 17.6.

FIGURE 17.24

Tertiary storage platforms: (A) Quantum tape library, (B) BluRay optical jukebox.

Table 17.6 Properties of Selected High-Capacity Tertiary Storage Systems

Manufacturer	Product	Maximum Capacity (TB)	Media Slots	Media Type	Interface	Power
Quantum	Scalar i6000 tape library	180,090	12,006	LTO-7 cartridge	8 Gbps Fibre Channel	24 kVA
HIT Storage	HMS-5175 BluRay library	175	1750	100 GB BDTL disc	1 Gbps Ethernet	

17.5 SUMMARY AND OUTCOMES OF CHAPTER 17

- Mass storage enables computational state retention to be consistent between power cycles of the machine.
- The majority of storage systems utilize four main types of mass-storage devices: HDDs, SSDs, magnetic tapes, and optical storage. Although they serve largely the same purpose, they substantially differ in the underlying physical phenomena used to implement data retention as well their operational characteristics and cost.
- The storage hierarchy is subdivided into four levels that differ in access latency and supported data bandwidth, with latencies increasing and effective transfer bandwidth dropping when moving away from the top level of the hierarchy.
- Primary storage comprises system memories, caches, and CPU register sets.
- Secondary storage is the first level that leverages mass-storage devices. Normally, CPUs cannot directly access the secondary (or higher-level) storage and therefore transfers of data between the primary and secondary storage have to be mediated by the OS and computer chipset.
- Secondary storage capacity grew 11 orders of magnitude between the 1940s and 2016. Device I/O bandwidth advanced six orders of magnitude over the same period.
- The most commonly used technology in the secondary storage tier is HDDs, which offer the industry's best cost per unit of storage coupled with satisfactory reliability.
- Redundant array of independent disks (RAID) attempts to address reliability issues of conventional mass-storage devices.
- Tertiary storage is distinguished from secondary storage in that it usually involves large collections of storage media or storage devices which are nominally in an inaccessible or powered-off state, but may be reasonably quickly enabled for online use.
- Tertiary storage comprises high-capacity data archives designed to incorporate vast numbers of removable media, such as tapes or optical discs.
- Latency remains one of the biggest performance bottlenecks plaguing most of the I/O devices in use today.
- SANs provide a block-level storage abstraction over a common network. Their purpose is to enable access to shared storage devices for multiple entities, including virtualized server pools or other hosts attached to a common network. The shared storage devices may include HDDs and SSDs, optical jukeboxes, and tape silos.

- Network attached storage is a common component of supercomputing installations. It provides centralized shared storage capability, frequently with a very large capacity, to multiple hosts, specifically including compute and login nodes.

17.6 QUESTIONS AND PROBLEMS

1. What are the main storage-related challenges presented by large HPC systems? Elaborate.
2. Identify parameters the values of which may be used to classify an arbitrary HDD to one of the market segment categories mentioned in the last column of Table 17.1.
3. Your product development team has been tasked with design and implementation of an in-flight entertainment system for a large airliner. Your responsibility is to select a suitable lightweight storage device from several technologies discussed in the chapter. Justify your choice taking into account (1) cost, (2) reliability of operation, (3) required storage capacity, and (4) performance. When considering your choices, be mindful of the target operating environment for the system.
4. A 4096-node cluster runs large-scale simulations that are checkpointed every 2 h using burst buffers for intermediate I/O storage. The nodes are equipped with 64 GB memory each and the ratio of nodes to burst buffers is 16:1. Calculate:
   a. the minimum required capacity of each burst buffer to keep the checkpoint phase as short as possible
   b. the duration of a large simulation before device failures appear, given that the TBW metric of each burst buffer is 400.
5. A RAID6 system uses eight 4 TB drives, including the minimum required number of parity drives. Calculate the effective read and write data throughput for the array. What is its effective data capacity? How many drives would be needed to assemble a RAID10 system of equivalent capacity? How would the data throughputs change?
6. SAN and NAS are similarly sounding acronyms that may confuse novices in the field. What are the differences and advantages of each over the other? Provide primary examples of their application.
7. A particle detector generates data streams requiring an aggregate bandwidth of 4 TB/s in bursts of up to 1 min long. The streams are analyzed by a 2048-node system that extracts events of interest and compresses them, reducing the data volume to 1/100th of the original size. The events of interest are then archived on a dedicated robotic tape storage using LTO-7 tapes at a sustained rate of 250 MB/s per deck. Given that experiments (each producing a single burst of data) are performed at 1 h intervals and the tape change overhead is factored into sustained storage bandwidth, answer the following.
   a. How many tape decks working in parallel are necessary to accommodate the extracted event data without forced interruptions to the experiment schedule or additional intermediate data buffers?
   b. If the capacity of a tape cartridge is 6 TB, how many tapes are required to provide data storage for experiments performed over the span of 1 year? What is the estimated shelf volume required to archive all cartridges written in 1 year if the dimensions of a single cartridge are 102 mm × 105 mm × 21 mm?

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 533

c. Assuming that data processing requires a negligible amount of memory in addition to that needed to hold the input data, how much DRAM (in powers of 2) must each node be equipped with to avoid the use of intermediate data buffers?

REFERENCES

[1] Internet2 Home, 2017 [Online]. Available: http://www.internet2.edu.
[2] Cray XC40 DataWarp I/O Applications Accelerator, Cray Inc., 2014 [Online]. Available: http://www.cray.com/sites/default/files/resources/CrayXC40-DataWarp.pdf.
[3] B. Alverson, E. Froese, L. Kaplan, D. Roweth, Cray XC Series Network, 2012. WP-Aries01−1112.
[4] IBM Corp., RAMAC, The First Magnetic Hard Disk, [Online]. Available: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/ramac/.
[5] IBM 62GV/STC 8800 Super Disk, Wiki foundry, [Online]. Available: http://chmhdd.wikifoundry.com/page/IBM+62GV+%2F+STC+8800+Super+Disk.
[6] Computer History Museum, Seagate ST-506, [Online]. Available: http://s3.computerhistory.org/groups/ds-seagate-st-506.pdf.
[7] G.D. Forney Jr., Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference, IEEE Transactions on Information Theory 18 (3) (1972).
[8] G. Binasch, P. Gruenberg, F. Saurenbach, W. Zinn, Enhanced magnetoresistance in layered magnetic structures with antiferromagnetic interlayer exchange, Physical Review B 39 (7) (1989) 4828−4830.
[9] A.J. Viterbi, Error bounds for convolutional codes and an asymptotically optimum decoding algorithm, IEEE Transactions on Information Theory 13 (2) (1967) 260−269.
[10] E. Eleftheriou, W. Hirt, Noise-predictive maximum-likelihood (NPML) detection for the magnetic recording channel, in: IEEE International Conference on Communications ICC'96, Dallas, TX, USA, 1996.
[11] International Committee for Information Technology Standards, 4.21 self-monitoring, analysis, and reporting technology (smart) feature set, in: Information Technology – AT Attachment 8-ATA/ATAPI Command Set (ATA8-ACS), T13/1699-D Revision 6a, American National Standards Institute, Washington, DC, USA, 2008.
[12] R.C. Bose, D.K. Ray-Chaudhuri, On a class of error correcting binary group codes, Information and Control 3 (1) (1960) 68−79.
[13] K.A. Immink, J.G. Nijboer, H. Ogawa, K. Odaka, Method of Coding Binary Data, USA Patent US 4501000 A, July 27, 1981.
[14] K. Odaka, Y. Sako, I. Iwamoto, T. Doi, L.B. Vries, Error Correctable Data Transmission Method, USA Patent US4413340 (A), November 1, 1983.
[15] The Library of Congress, National Institute of Standards and Technology, Optical Disc Longevity Study, Technology Administration, US Dept. of Commerce, 2007.
[16] H.M. Edwards, Galois Theory, Springer-Verlag, 1984.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 534

CHAPTER

FILE SYSTEMS

18

CHAPTER OUTLINE

18.1 Role and Function of File Systems **549**
18.2 The Essential POSIX File Interface **554**
18.2.1 System Calls for File Access 554
18.2.1.1 File Open and Close 554
18.2.1.2 Sequential Data Access 555
18.2.1.3 File Offset Manipulation 556
18.2.1.4 Data Access With Explicit Offset 556
18.2.1.5 File Length Adjustment 556
18.2.1.6 Synchronization With Storage Device 557
18.2.1.7 File Status Query 557
18.2.2 Buffered File I/O 559
18.2.2.1 File Open and Close 559
18.2.2.2 Sequential Data Access 560
18.2.2.3 Offset Update and Query 560
18.2.2.4 Buffer Flush 560
18.2.2.5 Conversion Between Streams and File Descriptors 561
18.3 Network File System **562**
18.4 General Parallel File System **565**
18.5 Lustre File System **569**
18.6 Summary and Outcomes of Chapter 18 .... **575**
18.7 Questions and Problems **576**
References **577**

18.1 ROLE AND FUNCTION OF FILE SYSTEMS

The mass-storage devices discussed in Chapter 17 use only a limited number of data-access interfaces. They are closely related to the low-level protocols used for data transfers between the device and the system's memory, and to the physical data layout and storage partitioning permitted by the device. The stored data may only be accessed at predefined granularity dependent on the particular device, namely at the level of physical *blocks* (occasionally also referred to as *records* or *sectors*) that typically vary in

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00018-6
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 535

size from 512 bytes to as much as 16 KB. Carrying out small modifications to stored data or appending information to it in smaller than block-sized amounts may require a sequence of multiple read and write operations. Such direct data accesses necessitate correct calculation of the physical addresses of relevant blocks. This computation in some cases may involve intrinsic parameters describing the storage geometry implemented by the device, such as the now-obsolete cylinder-head-sector schema utilized by older hard-disk drives. As many modern storage appliances in tandem with their operating system (OS) drivers attempt to hide the details of translation of logical block addresses to physical, device-specific locations as well as the remapping of damaged blocks, they provide only very limited means of keeping track of the allocation of the device's storage space or higher-level content management of utilized blocks. Furthermore, mass storage with asymmetric performance or protocol structures for different operation types, such as reading from and writing to optical discs, may require different strategies and scheduling to support these operations efficiently, placing the burden of encoding the related control algorithms and heuristics on the application writer. Even though direct access to the physical medium is on occasion necessary to extract a predictable level of performance from a storage device or ensure strict control of data state replication in some applications (virtual memory swap space, some database implementations), it is far too inconvenient for general use in multitasking and multiuser environments.

File systems provide an abstraction that addresses these issues and adds other usability and convenience features, including storage space management and organization, a consistent programming interface that is portable and mostly independent from the underlying mass-storage device types, and extensions that use functions of other system components through the same application programming interface (API). To achieve good performance and coordinate access to shared physical resources from multiple programs issued by multiple users, the file system code is usually integrated with the OS. As is explained in Section 18.2, certain elements of the file system programming interface may be also implemented at the runtime-system level, both to provide additional features and for efficiency. Additional storage is needed for *metadata*, or information that indicates various attributes of stored datasets, comprises file system data structures, and is used to manage the allocation of physical storage blocks. Because of that, the effective space available on a mass-storage device that implements a file system will typically be reduced by a few percent compared to its raw capacity. The most notable features commonly supported by file systems are as follows:

- *Organization.* The file system imposes a hierarchical layout using *directories* and *files* as its primary components. Directories serve as containers for other directories and files, while the files comprise the actual datasets written to or read from the mass storage. File systems rarely impose limitations on what kinds of information may be stored inside files; this is usually decided by the applications creating and accessing files. In many cases additional conventions and even software are required to decipher the actual contents of files, which otherwise may be viewed only as anonymous byte streams. Depending on the file system, file size is usually limited to a large value that rarely interferes with the practical aspects of file access. In most modern disk or solid-state device (SSD) file systems, that value is on the order of tens or hundreds of terabytes. Similarly, the directory space may have constraints, such as the maximum number of entries (files or directories) per directory it can accommodate, and possibly the total number of all directories and files coexisting in a single file system.

- *Namespace*. One of the most important usability aspects of a file system is the support of a naming scheme independent of the system architecture for stored information. All logical names of files and directories are expressed in the form of *paths*, or multicomponent strings in which each element names the containing directory ordered from the topmost in the hierarchy to the lowest level at the path's leaf component (which may be either a file or directory). Thus each file system component is uniquely identified by its symbolic name. While some details of path construction and how many roots, i.e., top-level hierarchy entry points, are supported differ between the individual file systems, the overall naming scheme conforms to the same common model across many implementations. One of the frequently used conventions to interpret the file contents relies on so-called *extensions*: a short suffix added to the file name and separated from it by an agreed character, typically a period. File system namespaces often support additional constructs, for example *links* that act as aliases for storable components. This permits the creation of alternative traversal paths that are not confined to the tree hierarchy and in some variants may even cross the file system boundary.
- *Metadata*. Due to the shared nature of file systems, access to certain datasets must be constrained to only preapproved users in the system. In Unix this is traditionally arranged at the level of the file or directory owner, a specific user group, and "others" (collectively all users known to the system). In each of these categories the access rights may be individually enabled or revoked for reading, writing, and executing a specific file system entry. Unix and compatible file systems may also specify additional flags, such as "sticky bits" (restricting file deletion only to the actual owner), *setuid* and *setgid* flags that elevate the effective execution privileges to those of the file's owner, or a flag that restricts the file's execution to specific users. Some implementations also support more fine-grained access schemes, such as access control lists (ACLs). They are more flexible than the default owner/group/others categories in that arbitrary permissions can be assigned to arbitrary users, albeit at the cost of additional space that may be required to store the list. Metadata are commonly used to describe other properties of files, most notably their size. Even though storage is allocated in blocks, file size is tracked with byte resolution (the last block of a file may be partially filled). File systems may combine several small files into a single block to conserve storage space. Note that large amounts of metadata remain opaque to the user, including the actual device block numbers allocated to the file as well as internal data structures that describe more complicated layouts, such as large files or files with "holes".
- *Programming interface (API)*. From the user's perspective, one of the fundamental operational properties of a file system is to permit creation of, writing to, and reading back the contents of files. This is accomplished through library calls that internally invoke the lower-level system functions. The files are identified by their symbolic names (paths) before the actual data access functions are enabled. This verifies that the target file exists and may be accessed by the requestor, and initializes the necessary data structures for access. It also relies on shorter and uniform file handles, eliminating the need to pass potentially highly variable file names to the data access functions. The API also allows specification of some metadata elements when new files are created. Besides file data access, the programming interface supports the manipulation of the storage hierarchy, such as directory traversal, file and subdirectory deletion, and creation of new subdirectories and links.

- *Storage space management*. As all physical devices have explicit capacity limits, the file system must carefully monitor the use of space in the storage medium, which may be shared potentially between millions of files and directories. Additionally, space for newly created files should be allocated in a way that ensures good access performance for the specific device type. Thus for standard hard-disk drives the file system typically strives to reserve the space for a file in continuous segments that reside on the same platter and cylinder, since sequential access offers the highest effective data bandwidth and latency. However, as the device becomes full, allocation in contiguous chunks may become more and more difficult (the available space becomes fragmented). Many modern file systems implement on-the-fly defragmentation algorithms, so performance degradation is not noticeable until the available capacity drops below a few percent (or even less in some cases). Other file systems may require explicit online or offline defragmentation to restore performance.
- *File system mounting*. Computers frequently utilize multiple storage devices at the same time. They can be made available for use at arbitrary points in time and not only during system initialization, as some of the storage media may be removable. This is performed in a process called *mounting*, in which the hierarchy defined by the imported file system is exposed to the OS and runtime environment. In single-namespace file systems utilized by some operating systems like Unix, this requires support to expand the existing name hierarchy. In such an OS the *mount points* under which the external file systems may be made accessible can conceivably be any existing directories. After the mount operation is completed, the imported file system hierarchy replaces the original layout extending below the mount point. Multiple file systems may be mounted at the same time, including nesting.
- *Special files*. The Unix environment is commonly known for implementing the "everything is a file" abstraction. This means the file system namespace may be used to provide access to other system entities and software constructs such as raw devices, named pipes, and sockets. The latter two enable interprocess communication as long as the interacting entities agree in advance on the name and type of the communication channel. While the communication uses the same API as that applied to transfer data to and from regular files, users must take care not to exceed the internal buffer capacity. Unfortunately, the elegance of the abstraction breaks when access to advanced features or adjustment of control parameters is necessary; in such cases the much-overloaded *fcntl* and *ioctl* interfaces are invoked to access the required functionality.
- *Fault handling*. As with any physical device, mass storage suffers from random failures. A properly designed file system can minimize the impact of these faults on stored data integrity. While device-related faults can range from individual bad blocks to whole devices, this does not exhaust the possible spectrum of failures. Due to data caching in memory and the need to perform multiple low-level updates even for a single logical access operation, commonly occurring problems are aborted write operations or destruction of unwritten data in memory caused by power fluctuations or system crashes due to other reasons. The data and/or metadata stored on disk may thus be left in inconsistent state and needing to be fixed before regular operation resumes. Many file systems deal with this by scanning the contents of data structures on the storage device during bootstrap and fixing incompatible entries using a dedicated utility program

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 538

(*fsck* in Unix). Such scan operations may be significantly accelerated when using an independent *journal*, or log of file system transactions that have to be carried out. While the file system check operation is not always able to recover all the data that were misplaced during a crash, it ensures that the loss is limited only to the data transferred during the failed operation and that stored metadata are consistent.

The landscape of currently available file systems covers many instances with different features and characteristics, deployment environments, target storage devices and media, and applications. There are file systems specifically optimized for use with hard-disk drives, SSDs, flash memories, tapes, and optical media. File systems may transparently support compression to save space and encryption to protect the confidentiality of the stored information. Pseudo file systems are used to expose details related to arbitrary installed devices and system data structures using familiar semantics (such as `procfs`, `sysfs`, and `devfs` in Linux). Particularly important to high performance computing (HPC) are distributed and parallel file systems, which support multiple clients communicating with storage devices over network or computer interconnects. However, unlike storage area networks (SANs), they do not share file contents at the physical block level but implement a service layer translating and executing received requests. Not all distributed file systems can necessarily provide high performance concurrent access to the same file from multiple clients, but instead focus on supporting the shared namespace and metadata and achieving significantly better throughputs when each client operates on its own disjoint set of files. This issue is better addressed by parallel file systems, making them more suitable for supercomputing applications which may read or write various sections of the same file or file set from multiple compute nodes. Note that this mode of operation is associated with several nontrivial challenges. Firstly, a parallel file system needs to employ appropriate mechanisms to accommodate multiple storage devices by distributing the contents of files over multiple disks or SSDs (striping). This is necessary to extract the required aggregate data throughput. The stripe unit has to be carefully chosen so as not to impose too high an overhead (small blocks) and not to destroy striping benefits for smaller files (large blocks). Secondly, the file system is expected to provide the abstraction of a single server to accessors: details of the underlying architecture, physical arrangement of supporting hosts and storage devices, fault-tolerance measures, file striping parameters, and many other aspects should be hidden from users who are not interested in optimizing the input/output (I/O) performance for specific applications. A familiar file access interface (such as Portable Operating System Interface or POSIX) may be provided to reduce the learning curve for new users and facilitate application porting. Thirdly, both metadata and data have to utilize appropriate consistency protocols, since there must be no discrepancies between file contents, sizes, and other attributes when simultaneously viewed by different nodes. While multiple readers of a file can easily be accommodated, the addition of even a single writer may complicate the way information is propagated to and possibly replicated on the participating nodes. Parallel file systems may also resort to relaxing access atomicity (i.e., the guarantee that no portion of data read or written in a single call is ever modified by an overlapping preceding or subsequent access) to be able to attain reasonable data throughput rates. Finally, the governing algorithms must scale to support not only a large number of concurrent accessors, possibly extending to the total number of compute nodes in the system, but also growing storage pools.

18.2 THE ESSENTIAL POSIX FILE INTERFACE

The POSIX standard [1] describes the elements of runtime API, shells, and utilities, specifying compatibility requirements for variants of the Unix operating system. The file I/O interface is part of the specification. The necessarily limited overview presented here focuses only on a subset of data transfer functions, with a few auxiliary calls that are frequently used in parallel programs. Directory access and manipulation, link creation, file deletion, and other namespace and metadata functions are not discussed, as they are rarely invoked directly from applications but instead are typically handled by job scripts using appropriate system utilities. File access functions come in two flavors: system calls and buffered I/O. Both are described below, along with usage examples and enumeration of their semantic differences.

18.2.1 SYSTEM CALLS FOR FILE ACCESS

System calls are used to invoke OS kernel functions directly. While all system calls typically share the same generic invocation format, a thin wrapper layer is additionally provided by the runtime library for user convenience and to facilitate first-level argument checking. Since system calls incur greater overheads than regular user—space function invocation, this interface should be used to transfer larger amounts of data (several memory pages or more) per call. The interfaces described below show function arguments and the necessary "include" files, defining their prototypes and optional argument macros. Since system calls are often used to access other entities in the system, such as terminals, pipes, or sockets, only semantics related to regular file access are discussed here.

18.2.1.1 File Open and Close

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int flags, ...);
```

```
#include <unistd.h>

int close(int fd);
```

The `open` call allocates an integer *file descriptor* that shall be used in all subsequent accesses to the file whose name is specified in the `path`. The descriptor returned is the lowest integer not currently used for file access by the calling process, and identifies the kernel data structure associated with the opened file. The `flags` argument consists of just one of `O_RDONLY`, `O_WRONLY`, and `O_RDWR` for read-only, write-only, and mixed read-and-write access respectively. The access mode flag can be bitwise or-ed with the arbitrary combinations of flags listed below.

- `O_APPEND` causes initial file offset to be set to the end of file instead of its beginning.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 540

- O_CREAT creates the file if it does not exist, and is otherwise ignored as long as O_EXCL is not set. The file is created with access rights specified in the third argument that conform to conventional owner/group/other permissions.
- O_EXCL when used together with O_CREAT will cause the call to fail if the file exists. If the flag is specified without O_CREAT, the result is undefined.
- O_TRUNC truncates the existing file to zero length if the access mode is O_WRONLY or O_RDWR. Using this flag in read-only mode produces an undefined result.

The list of supported flags in the open call is fairly extensive and permits among other uses the specification of nonblocking accesses and synchronization of write operations. The description of their exact semantics is beyond the scope of this brief overview.

A successful open call returns a nonnegative integer that is a valid file descriptor. A negative one is returned on failure and a corresponding code is set in the global errno variable. Error causes include insufficient access or file creation rights, invalid path, exceeded maximum number of simultaneously opened files in the system, and requested file creation with an exclusive flag but the target file already exists. A failed open cannot modify an existing file status or create a new file.

The opened files may be closed by passing their descriptors to the close call. This causes deallocation of the file data structure and releases the file descriptor for reuse within the calling process.

18.2.1.2 Sequential Data Access

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t n);
ssize_t write(int fd, void *buf, size_t n);
```

The read function attempts to read at most n bytes at the current offset from the file identified by fd into a user buffer pointed to by buf. Successful invocation returns the actual number of bytes stored in the user buffer. The call may return a value less than n if the number of bytes between the current offset associated with fd and the end of the file is smaller than the requested value. Partial read interrupted by a signal may also return fewer bytes than requested. A successful call will increase the file offset by the number of bytes transferred to the user buffer and update the file access time to the system time at which the access was carried out.

A negative return value indicates an error whose cause is identified in the global variable errno. Possible error causes include the use of an invalid file descriptor, exceeding the maximum offset, and a read operation that has been interrupted by a signal without having started yet.

The write call attempts to transfer n bytes provided in the user buffer pointed to by buf to a file identified by the descriptor fd. The position at which the data are stored in the file is determined by the current value of the file offset associated with the descriptor. If the offset of the last written byte is greater than the file length, the file length will be updated to the position of the last written byte increased by one. A successful call returns the actual number of bytes written; the internal file offset is incremented by this value and file's modification and status timestamps are updated as well. If the write would exceed the maximum file size limit or medium capacity, only the portion of user buffer that can

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 541

be accommodated is written. Successful file updates are immediately visible to other accessors (including other processes); "read" from file locations affected by a successful write call will return the data transferred by that call. This data will persist only as long as there is no subsequent write call issued that would overwrite the data in the same position.

Similarly to "read", the write function returns −1 on error. The error causes resemble those of "read", with the addition of writes that would exceed the maximum file size without the possibility of performing a partial data transfer.

18.2.1.3 File Offset Manipulation

```
#include <unistd.h>

off_t lseek(int fd, off_t offs, int whence);
```

The lseek call is used to modify the file offset associated with the descriptor fd. The semantics of the call depend on the value of the whence parameter. The offset is directly set to offs value if whence is SEEK_SET. If whence is SEEK_CUR, the file offset is set to the sum of the current offset value and offs. Finally, for SEEK_END the resultant offset is the length of file plus offs. Note that the file offset can be advanced to point beyond the end of file; the unwritten segments of the file will read as zeros until they are overwritten. The call returns the updated offset value (measured from the beginning of the file) in bytes.

18.2.1.4 Data Access With Explicit Offset

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t n, off_t offs);
ssize_t pwrite(int fd, void *buf, size_t n, off_t offs);
```

The pread and pwrite calls provide explicit offset variants of the read and write functions. They save the explicit invocation of lseek when accesses at random locations in the file need to be performed. The value of the implicit file offset associated with the descriptor fd is not modified by the calls.

18.2.1.5 File Length Adjustment

```
#include <unistd.h>

int ftruncate(int fd, off_t len);
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 542

The `ftruncate` function sets the length of file identified by `fd` to `len`; the file must be opened for writing. The result may be the effective truncation of file length, in which case the data located at and after `len` offset will no longer be accessible to reads or file length increase, with the appended data segment reading as zero-filled. The `ftruncate` function does not modify the value of the file pointer associated with the descriptor `fd`.
The call returns zero on success or −1 on error.

18.2.1.6 Synchronization With Storage Device

```
#include <unistd.h>

int fsync(int fd);
```

The `fsync` function transfers all data and metadata associated with the file identified by `fd` to the underlying storage device. The call blocks until all data are transferred or an error occurs. On success zero is returned, otherwise it is −1.

18.2.1.7 File Status Query

```
#include <fcntl.h>
#include <sys/stat.h>

int lstat(const char *restrict path, struct stat *restrict buf);
int fstat(int fd, struct stat *restrict buf);
```

Both calls retrieve metadata of the file system entity identified either by `path` (`lstat`) or by the opened file descriptor `fd` (`fstat`) in a status structure pointed to by `buf`. They return a value of zero on success and −1 otherwise. Individual metadata entries are stored in different fields of `struct stat`, and include among others:

- **st_size**—size of file in bytes
- **st_blksize**—size of block used by file system in I/O operations
- **st_mode**—file type "sand" mode; if set, bit flags S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH identify enabled read and write access rights for user, group, and others in the system
- **st_uid**—user ID of file owner
- **st_gid**—group ID of file owner
- **st_atim**—time of last file access
- **st_mtim**—time of last modification of the file
- **st_ctim**—last status change time.

Code 18.1 shows an example code using a system call file interface to write a number of integers to a created (or truncated) file, flush it to persistent storage, and read back a smaller section of written file.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 543

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6
7 #define BUFFER_SIZE 4096
8 #define HALF (BUFFER_SIZE/2)
9
10 int main(int argc, char **argv)
12 {
12    // initialize buffer
13    int wbuf[BUFFER_SIZE], i;
14    for (i = 0; i < BUFFER_SIZE; i++) wbuf[i] = 2*i+1;
15
16    // open file, write buffer contents, and flush it to the storage
17    // the file is accessible (read/write) only to the creator
18    int fd = open("test_file.dat", O_WRONLY | O_CREAT | O_TRUNC, 0600);
19    int bytes = BUFFER_SIZE*sizeof(int);
20    if (write(fd, wbuf, bytes) != bytes) {
21      fprintf(stderr, "Error: truncated write, exiting!\n");
22      exit(1);
23    }
24    fsync(fd);
25    close(fd);
26
27    // retrieve the second half of the file and verify its correctness
28    int rbuf[HALF];
29    fd = open("test_file.dat", O_RDONLY);
30    bytes /= 2;
31    if (pread(fd, rbuf, bytes, bytes) != bytes) {
32      fprintf(stderr, "Error: truncated read, exiting!\n");
33      exit(1);
34    }
35    close(fd);
36
37    for (i = 0; i < HALF; i++)
38      if (wbuf[i+HALF] != rbuf[i]) {
39        fprintf(stderr, "Error: retrieved data is invalid!\n");
40        exit(2);
41    }
42    printf("Data verified.\n");
43
44    return 0;
45    }
```

Code 18.1. Example demonstrating the use of I/O system calls to create, write, and read data from a file.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 544

18.2.2 BUFFERED FILE I/O

Buffered file access is implemented by the Unix runtime system library, `libc`. It introduces additional data buffers in the application's address space that may improve performance if frequent operations involving small amounts of data are performed. The buffers and their control parameters are not exposed directly to the application. Whenever possible, I/O calls issued by users are satisfied by copying the data between the user buffer in the application and the internal library buffer, thus avoiding the overhead of system calls. Occasionally system calls have to be issued to access the underlying physical storage, but their cost is amortized by transferring large amounts of data between the OS kernel and library buffers either by performing read-ahead for the input stream or waiting until the internal buffer is sufficiently filled before handing it off to the kernel. This interface is also known as the *streaming* interface (and related file description structures as *streams*), since the best performance is achieved during sequential access. As the buffering layer is not exposed to the kernel, the newly written file data may not be immediately visible to other accessors of the file in the system and are also more likely to be lost in a system crash.

This interface is part of the *stdio.h* chapter of the International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) C language standard [2] and is thus far more portable than functions based on system calls.

18.2.2.1 File Open and Close

```
#include <stdio.h>

FILE *fopen(const char *restrict path, const char *restrict mode);
int fclose(FILE *stream);
```

The `fopen` call opens or creates a file identified by `path` and associates it with a stream. The first character of the mode argument determines the file access mode and may be one of the following:

- "r" opens the file for reading
- "w" creates a file or truncates the file to zero length if it already exists and opens it for writing
- "a" creates or opens a file for write access at the end of file (append mode).

The mode string may also contain a "+" character which enables access in *update* mode, or both reading and writing performed in any order. The other characteristics defined by the first character of the mode string are preserved. If the file is used in update mode, the application must ensure that input and output operations are separated by a seek call, or, in case of reads following writes, `fflush`.

A successful call to `fopen` returns a valid stream pointer, or NULL otherwise.

The opened streams may be closed using the `fclose` function. The side-effect of close operation is propagation of the contents of data buffers to the file. Invocation of `fclose` causes the `stream` to be disassociated from the underlying file independently of return status. The function returns zero on success or EOF on failure. Common error causes include exceeding the file size or offset limit while attempting to flush the buffer contents to storage, exhausting the space available on the device, and receiving a signal while executing `fclose`.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 545

18.2.2.2 Sequential Data Access

```
#include <stdio.h>

size_t fread(void *restrict buf, size_t size, size_t n, FILE *restrict stream);
size_t fwrite(const void *restrict buf, size_t size, size_t n, FILE *restrict stream);
```

The fread and fwrite functions are stream equivalents of read and write calls. They attempt respectively to read or write an integral number of elements, n, each of size size bytes, from an opened stream stream by transferring them from or to the user buffer pointed to by buf. Both functions return the number of elements successfully transferred. The return value may be less than n only if the end of the file has been encountered while reading or an error has occurred during writing. The file offset associated with the stream is increased by the number of bytes successfully transferred. If an error occurs, the value of offset for the file associated with the stream is unspecified.

18.2.2.3 Offset Update and Query

```
#include <stdio.h>

int fseek(FILE *stream, long offs, int whence);
long ftell(FILE *stream);
```

The fseek function sets the value of the file offset for a specified stream in accordance with the values of offs and whence arguments. The latter can be one of SEEK_SET, SEEK_CUR, and SEEK_END; their interpretation is the same as for lseek. Upon success, fseek returns zero or −1 on error. fseek causes the yet-unwritten buffered data to be propagated to the underlying file.
The ftell call returns the current value of the internal file offset associated with stream stream measured in bytes from the start of the file. The error is indicated by −1 as a return value. Note that ftell fails if the current offset cannot be correctly stored in a variable of long type.

18.2.2.4 Buffer Flush

```
#include <stdio.h>

int fflush(FILE *stream);
```

The fflush function forces the unwritten data stored in a buffer associated with a stream opened in write or update mode to be written to the underlying file. If the stream has been opened for reading, the call will set the offset of the underlying file to the current offset position of the stream. If the stream is a null pointer, the function will perform the described action for all opened streams. The call returns zero on success and EOF on error.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 546

18.2.2.5 Conversion Between Streams and File Descriptors

```
#include <stdio.h>

FILE *fdopen(int fd, const char *mode);
```

```
#include <unistd.h>

int fileno(FILE *stream);
```

On occasion it may be useful to convert between streams and file descriptors to be able to invoke alternative interface functions. For example, the stream library does not provide any calls to force data propagation to the physical storage medium; this is typically handled by kernel functions. Similarly, switching to a buffered interface may be beneficial if large numbers of fragmented sequential I/O operations are to be carried out. Thus `fdopen` accepts an open file descriptor and mode string whose meaning is the same as for the `fopen` call, and creates and returns a corresponding stream descriptor. The supplied `mode` argument has to be compatible with the access mode of the file referred to by the descriptor `fd`. The offset of the returned stream will be set to the same value as that of the opened file indicated by `fd`. A failed call returns a null pointer.
The converse operation, `fileno`, extracts the descriptor of the underlying file from the specified stream structure, or returns −1 to indicate an error.

Code 18.2 presents a converted version of a program originally listed in Code 18.1 that uses a buffered I/O interface instead of system calls. While the transformation is obvious for most I/O functions used, one detail is particularly noteworthy. Since the `fflush` call native to the `stdio` library can only push the contents of stream buffers to the kernel, the actual propagation of dirty data to storage has to be performed by a system call (`fsync`). To provide the file descriptor expected as an input argument to that call, `fileno` is used to retrieve it from the stream descriptor (line 21).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 #define BUFFER_SIZE 4096
6 #define HALF (BUFFER_SIZE/2)
7
8 int main(int argc, char **argv)
9 {
10  // initialize buffer
11  int wbuf[BUFFER_SIZE], i;
12  for (i = 0; i < BUFFER_SIZE; i++) wbuf[i] = 2*i+1;
13
14  // open file, write buffer contents, and flush it to the storage
15  FILE *f = fopen("test_file.dat", "w");
```

```
16    size_t count = BUFFER_SIZE;
17    if (fwrite(wbuf, sizeof(int), count, f) != count) {
18       fprintf(stderr, "Error: truncated write, exiting!\n");
19       exit(1);
20    }
21    fflush(f); fsync(fileno(f));
22    fclose(f);
23
24    // retrieve the second half of the file and verify its correctness
25    int rbuf[HALF];
26    f = fopen("test_file.dat", "r");
27    count /= 2;
28    fseek(f, count*sizeof(int), SEEK_SET);
29    if (fread(rbuf, sizeof(int), count, f) != count) {
30       fprintf(stderr, "Error: truncated read, exiting!\n");
31       exit(1);
33    }
34    fclose(f);
35
36    for (i = 0; i < HALF; i++)
37    if (wbuf[i+HALF] != rbuf[i]) {
38       fprintf(stderr, "Error: retrieved data invalid!\n");
39       exit(2);
40    }
41    printf("Data verified.\n");
42
43    return 0;
44 }
```

Code 18.2. Equivalent program to Code 18.1 that uses the streaming I/O interface.

18.3 NETWORK FILE SYSTEM

The Network File System (NFS) is one of the oldest and at the same time one of the most broadly deployed distributed file systems in computing installations. Originally conceived at Sun Microsystems in 1984, it is currently an open standard that has spurred many implementations, including several open-source versions. Its main appeal is that a regular file system with access confined to a single host can be "exported" to permit remote access to its contents (files, directories, links, etc.) from multiple client machines. There are no significant restrictions regarding the properties of the underlying file system; any POSIX-compliant file system can be accessed via NFS and in some cases (e.g., new technology file system through the Microsoft Subsystem for Unix-based Applications) even file systems with incompatible interfaces are available. The remote file system can be transparently mounted at any place in the directory hierarchy and accessed as if it was local. Earlier revisions of NFS were frequently described as *stateless* protocols, since the server did not track clients which mounted the file system or which files were in use. This has the benefit of easy recovery after failures: the client must only retry the request until the server responds, but without renegotiating the connection and causing rebuild of the

preexisting state or generating a new, incompatible state. While some persistent data structures had to be introduced to alleviate certain problems, the protocol attempts to limit the additional server-side state as much as possible. The NFS requests are self-contained, which makes the protocol very efficient.

NFS services can utilize both transmission control protocol (TCP) (connection-oriented) and user datagram protocol (UDP) (datagram based) messages. At the heart of the protocol stack is support for Remote Procedure Call (RPC), which permits sending requests from clients to a remote host, invocation of a function local to the host, and propagation of returned data and operation status in reply packets. Originally based on Sun RPC implementation, it is now defined by the Open Network Computing (ONC) RPC specification [3]. RPC implementation must uniquely specify the procedure to be called on the remote end, match the response messages to original requests, and define provisions for authenticating the requestor to service and vice versa. It also handles errors caused by protocol and version mismatch, unavailability of the requested procedure on the server, and authentication failures. Due to the requirement to support hosts with different data type properties and byte order, an external data representation [4] layer is used to serialize and retrieve the call arguments and other data that are conveyed as packet payloads. To support RPC, *port mapper* services on a dedicated port 111 must be configured on the participating machines. ONC RPC was relicensed in 2009 to use the standard three-clause Berkeley Software Distribution license.

The basic architecture of NFS is illustrated in Fig. 18.1. Before users are permitted to issue any data access requests, the remote file system has to be mounted on the client host. This is accomplished by the mount program parsing the name of the NFS server and asking it to provide the handle for the

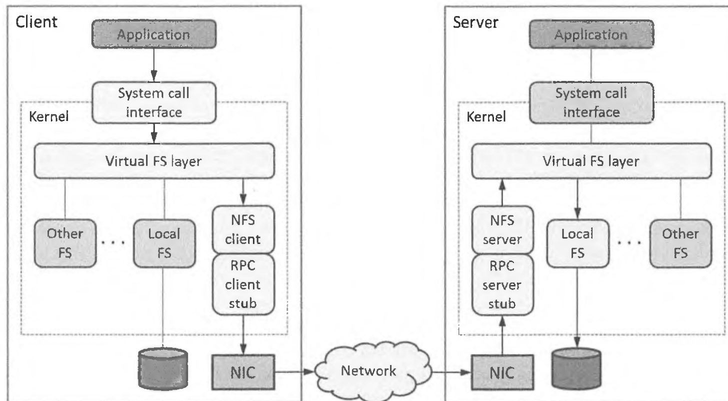


FIGURE 18.1

Architecture of Network File System and its integration with other kernel components in Linux. The arrows show the propagation of client requests to the server and remote file system. The virtual file system (VFS) layer provides an implementation-independent interface to access the underlying file system(s). The NFS client relies on the Remote Procedure Call (RPC) service to enable transparent invocation of file system functions on a remote node as requested by a client.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 549

remote directory. If the requested directory exists and export is permitted, the server returns its handle. This causes the local kernel to access the virtual file system (VFS) layer and create a virtual node (vnode), or translation from a symbolic path to an arbitrary accessed file system object, for the remote directory. Among other things, vnodes store information about whether the target object is local or remote. Thus the subsequent open request for the remote file issued by the user finds the parent portion of the file's path that translates to vnode marked as remote, retrieves the stored server address, sends the *lookup* request to the server utilizing the RPC code stubs on client and server, and creates the opened file entry using the retrieved file attributes provided by the server. The corresponding descriptor index is then returned to the user program. The lookup procedure is used since the server does not execute a regular `open` call to avoid creation of state; as a result of using lookup a specially formed *handle* is returned that uniquely identifies the file to the server. Data access, such as read operations, proceeds similarly, except that since the client may be permitted to cache the file data locally in newer NFS revisions, a local cache lookup is performed to check if the data are available locally. NFS servers also use a simple strategy to deal with request duplication, such as that caused by packet retransmits due to network errors. This applies only to *nonidempotent* requests, i.e., those that would fail if retransmitted, such as directory or file removal. The servers maintain a *request replay cache* in which all nonidempotent requests are kept for a predetermined period; finding that a newly received request's transaction ID, source address, and port match one already in the cache will suppress its execution and cause the cached reply to be reemitted.

The first publicly released version of NFS was version 2 (NFSv2). Since it was developed in the late 1980s, it is considered dated by today's standards. For example, NFSv2 used 32-bit signed integers for file offsets, practically limiting access to the first 2 GB (gigabytes) of data per file. The size of the data payload per packet was limited to 8 KB; this, coupled with synchronous operation in which the server must complete a data write before issuing a reply to the client, caused poor write throughput. While asynchronous operation was possible, it gave rise to silent corruption of data in certain circumstances. Another problem of NFSv2 was lack of data consistency enforcement across multiple clients. File handles in this version were 32 bytes long.

NFSv3 was a much-improved revision of the protocol that still preserved the "stateless" design. It is still found in use today, although many data centers and institutions switched to the next version, which introduced some minimal state at the server to handle features that otherwise would have to be supported externally. Version 3 offered 64-bit offsets, practically removing file size limitations. The per-packet payloads increased to about 60 KB for UDP and typically 32 KB with TCP. A weak cache consistency scheme was implemented to detect changes to files made by other clients. This was achieved by injecting current file attributes into the server's reply to read and write requests; these could be used by the client to determine if its cached file data or attributes were stale. If this was the case, the client would discard the cached information and flush any dirty data to the server. While NFSv2 clients were interpreting mode flags passed to the `open` call directly to verify access permissions, NFSv3 made this a server's responsibility (using the `access` call), thus enabling correct access to file systems supporting ACLs from non-ACL-aware clients. The write performance was also improved by storing data sent in multiple data requests (while acknowledging each received packet) in memory and then committing all of them at once to disk.

The current revision of NFS was heavily influenced by the design of the Andrew File System [5] and Microsoft's Common Internet File System (CIFS) [6]. NFSv4 supports operations that inherently require server-side state, such as file locking. The new protocol is capable of byte-range locking that is

lease based. Since clients may crash before releasing active locks, it forces them to stay in touch with the server for the duration of locked operations. Otherwise, the locks are revoked after preset timeout. A new approach to caching of file contents called *delegation* has been introduced. It permits a client to modify files locally in its own cache without communication with the server. Read delegation can be granted to multiple clients simultaneously, while write delegation may be permitted to only one client at a time. When a conflict is detected for the currently held delegation(s), they may be revoked using a callback mechanism. Version 4 improves overall response time by permitting compound RPCs, i.e., calls that combine several commonly executed request sequences (such as lookup, open, and read) into one. The security of operation and authentication has been substantially augmented through introduction of Kerberos 5 [7] and SPKM/LIPKEY [8]. The administrative overhead required to coordinate numeric user and group IDs across multiple hosts and to enforce conventional Unix permission flags is reduced thanks to the new ACL mechanism that interoperates with both POSIX (though not perfectly) and Windows ACLs, with user names expressed as strings. Finally, the NFSv4 protocol implements file migration and replication.

Despite these improvements, NFS best supports *session semantics*, in which clients have exclusive access to files and the updates to them are propagated on file close (session finish). Scenarios where multiple applications perform modifications of a shared file, such as appending to a shared log file, will not achieve good performance. While the optional parallel NFS extension introduced in the minor revision 4.1 [9] of the standard supports rudimentary parallel access semantics, these operations are better left to parallel file systems, two examples of which are discussed in the next sections.

18.4 GENERAL PARALLEL FILE SYSTEM

The General Parallel File System (GPFS) was developed by IBM and released commercially in the late 1990s. Its functionality has been influenced by the Tiger Shark file system [10] research project at IBM Almaden, oriented to provide high performance multimedia streaming. GPFS also incorporates design ideas from an earlier Vesta parallel file system designed by IBM [11]. It supports concurrent access from multiple clients to possibly multiple file system instances distributed over physical storage devices in the system. The storage devices can be either accessible via SAN or exported over network using higher-level protocols. The file placement optimizer is a feature that allows efficient GPFS operation in the “shared-nothing” cluster architecture frequently favored by “big data” applications. GPFS features data replication, providing high recoverability and availability, policy-based storage management, a global namespace that permits shared file access across different GPFS instances (called GPFS *clusters*) on wide area networks (WANs), and standard (including POSIX) file interfaces that support conventional OS file system utilities as well as execution of unmodified applications. Similarly to NFS, the latter is accomplished through kernel extensions that inject GPFS functionality into the VFS layer, making it appear to the kernel as another natively supported file system. The high level of performance is achieved by spreading data accesses across multiple storage devices (to obtain high aggregate data bandwidth), load balancing to eliminate storage hotspots, efficient support for concurrent reads and writes from multiple clients (even to the same files), a sophisticated token management system as a basis for distributed lock management and file data consistency, intelligent prefetching of file data recognizing sequential (forward and reverse) and various forms of strided I/O patterns, and the ability to specify multiple networks for communication between GPFS daemons.

Table 18.1 Selected Operational Parameter Limits of the Current Version of GPFS

Parameter	Design Limit	Tested Value
Number of joined nodes per cluster	16,384	9,620
Number of disks per cluster	2,048	Unknown
File size	2^{99} bytes	Approximately 18 PB
Number of files per file system	2^{64}	9,000,000,000

GPFS implementation of journaling (I/O transaction logging) improves the chances of recovery after system crashes. The architectural limits of the main operational parameters listed in Table 18.1 give an idea of the extent of scaling supported by GPFS. The most recent revision, GPFS v4.2, is available for AIX (Power processor), Windows, and Linux OS (x86 series processors). Since 2015 the IBM GPFS brand has been known as IBM Spectrum Scale.

The basic architecture of GPFS is illustrated in Fig. 18.2. The diagram shows two configurations, one with I/O nodes resembling a traditional network attached storage arrangement and separated from

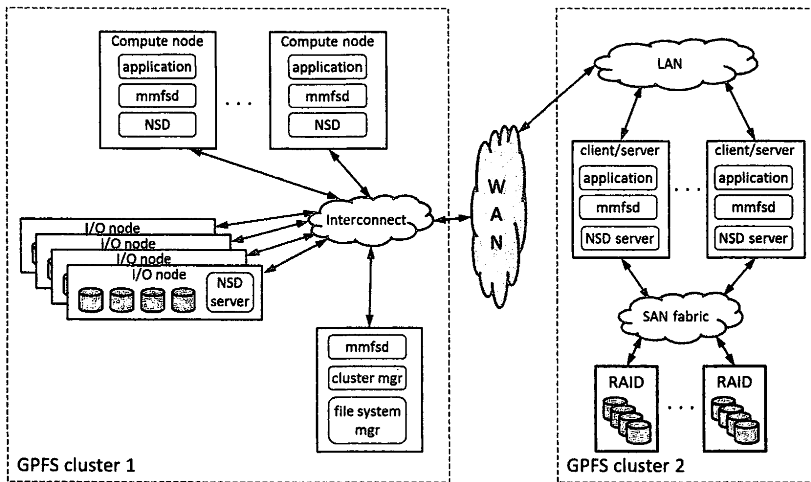


FIGURE 18.2

Some of the possible deployment configurations of GPFS: a network attached storage pool servicing a collection of client compute nodes on the left (GPFS cluster 1), and a server group managing SAN devices based on redundant array of independent disks on the right (GPFS cluster 2). Both clusters may communicate thanks to the shared WAN connection. Physical storage in both installations is abstracted through the Network Shared Disk (NSD) protocol. The core file system functionality is provided by the GPFS daemon, mmfsd, distributed across multiple computational resources in the system.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 552

the compute nodes, and a second with nodes that provide storage server capabilities while also permitting client applications to run. The second configuration transparently integrates a SAN storage pool which may offer enhanced resilience in case of either disk or network link failures through redundant links exposed by the SAN fabric. Other configurations are also possible. Both GPFS instances (clusters) may interact thanks to the WAN connection. The storage devices in GPFS installations are abstracted via the Network Shared Disk (NSD) protocol. They provide cluster-wide naming and high-bandwidth access to disk data for all clients that have no physical access to the underlying storage. NSD *servers* are started on the storage-equipped nodes, thus exposing virtual storage connections to other NSD components. For robust resilience, each NSD component may be associated with up to eight NSD servers: if one server fails, the next one in the list takes over. Older revisions of GPFS executing on the IBM SP series of machines provided an analogous service using virtual shared disk entities that communicated using a proprietary IBM interconnect. The current NSDs relax this constraint to permit other network types, but still require that a high-speed network is present.

GPFS daemons (denoted *mmfsd* in the figure) implement the core functionality of GPFS, including support for all I/O operations and data buffer management. They are instantiated as multithreaded processes with a separate group of dedicated threads for high-priority requests. Multiple daemons may communicate with each other to coordinate changes in configuration and recovery, and to synchronize concurrent updates to the same data. GPFS daemons are responsible for allocation of disk space required by newly created files and when existing files need to be extended; management of directories, including creation of new directories, updating the contents of existing directories, and identification of directories with pending I/O operations; lock management to protect the integrity of both file data and metadata; starting the related I/O operations; and quota accounting. To optimize performance, the daemons take advantage of *pagepool*, a pinned memory region that contains data and metadata of selected files. It is used to support frequent writes that may be overlapped with the execution of applications and data that are frequently reused (but fit into pagepool), and to provide buffer space for data prefetch, thus accelerating the performance of large sequential reads. Nonpinned memory may be allocated from the kernel heap and is primarily used to hold control structures and vnode information related to in-kernel aspects of file system management. The in-daemon shared memory is used as *inode cache* (inodes, short for index-nodes, are internal data structures used by the file system to control file layout) and *stat cache* that contains a subset of attributes of the most recently accessed files and directories. The daemons may also allocate internal nonshared memory segments to support operation of file system manager functions (including token management).

The file system manager (one per file system, but possibly distributed across multiple nodes), which may run on a dedicated node or as part of a regular client node, supervises the operation of all nodes using the file system. It provides services oriented on file system configuration (expanding the storage pool, performing file system repairs, and adjustment of disk availability), storage space allocation, token management, and quota management. Token management is critical to concurrent operations performed on shared GPFS files. If the file system manager executes on multiple nodes, the load is distributed across all participating token management servers. Token services issue tokens that temporarily grant file access rights (read and write of file data and metadata) to token holders. This locking is done per byte range, thus permitting simultaneous read accesses to some portions of a file while enforcing a rigorous order of updates on portions of file that are targeted by writes without explicit serialization of all requests. Interaction with the token server happens the first time a node

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 553

requests access to a file. After having been granted a read or write token, the client may perform compatible data accesses without further contact with the token manager. If the token server detects a conflicting access, it provides a list of all nodes holding tokens to the requested byte range. To avoid blocking the token server, it is the requesting client's responsibility to get the current token holders to relinquish them. As this must potentially wait for release of locks held on file, often the related pending I/O operations must be completed.

Each GPFS cluster has one associated cluster manager, elected by a quorum of the nodes constituting the cluster. The cluster manager keeps track of disk leases, monitors node failures and supervises recovery processes while ensuring that the necessary quorum of nodes exists to support the continued operation of the cluster, propagates configuration changes to remote nodes, chooses the file system manager node(s), and performs user identifier mapping from remote nodes.

Since concurrent write operations are often the source of conflicts when performed on shared files, it is educational to analyze the involved data and metadata paths (Fig. 18.3). A dirty block of data must be written when a system command requesting a flush of buffered data to storage has been invoked, a write in synchronous mode was called, the system needs to reuse buffers currently occupied by dirty data, a file token has been revoked, or the last byte of file block accessed sequentially has been written. Each open file in GPFS is associated with precisely one *metanode*, which is used to maintain metadata integrity. Typically the metanode is located on the host that had the file open for the longest period of

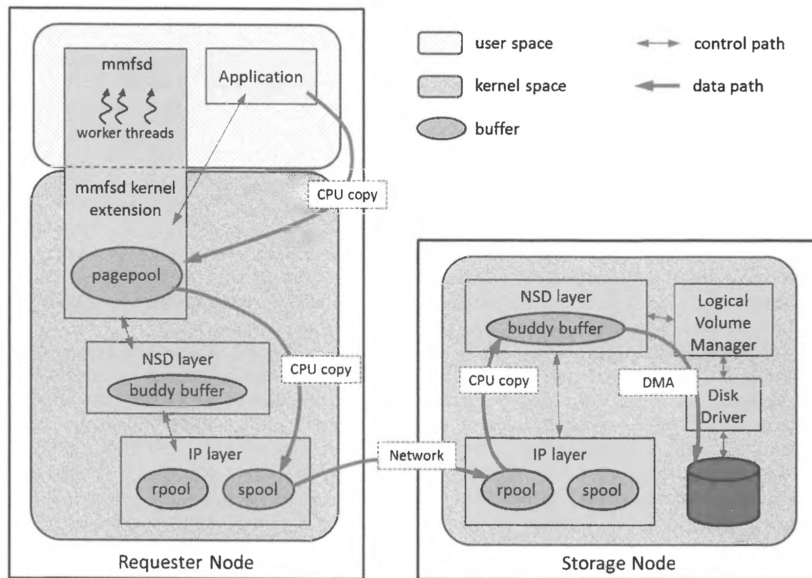


FIGURE 18.3

Data and control paths for execution of a remote write in GPFS.

time and functions as a synchronization point for file metadata for all nodes in the system. Both data and metadata are flushed as described in the following three scenarios of varying complexity:

1. *Buffer available in client memory.* This occurs if the buffer has been created for a previous write and the write token is still available. The contents of the application buffer are copied to the GPFS data buffer; at this point the write is complete from the application perspective. If the buffer flush conditions listed above are fulfilled, the daemon schedules an asynchronous buffer write to storage using one of its threads. This permits the write to overlap with application execution. The GPFS worker thread calls the NSD layer, causing the request to be broken up into chunks fitting message payloads and copied to the communication buffers in send pool. The list of destination I/O nodes is derived from file metadata. The data are transferred over the interconnect to the NSD server receive buffer pool. As soon as all packets are received, a buddy buffer is allocated to reassemble the write buffer contents. At this stage the related receive buffers in the NSD server are released and disk write is initiated. The latter may be delayed by a preconfigured time to permit coalescing with other neighboring write requests. Since the buddy buffer may not always be available, the request could remain queued with data stored in receive pool buffers until sufficient space is provided.
2. *Write token locally available, but data buffer absent.* This may happen if the buffer has been reused due to recent I/O activity or a previous write did not "touch" all data locations for which it obtained a token. Kernel code suspends the calling thread and instructs a daemon thread to obtain a buffer. If the write range covers the whole block (full overwrite), a new empty buffer is allocated. If the write affects a portion of a block and the remainder of the block exists, the remaining portion of the block is fetched and placed in the buffer. The call then proceeds as described in (1).
3. *Both data buffer and token are unavailable.* First a token for a specific byte range must be acquired. Based on the discovered I/O pattern, the byte range may be larger than the one requested by the application in anticipation of future requests, as long as no conflicts are detected with other accessors of the file. The token management may be forced to revoke the token currently owned by another node. After the token is obtained, processing progresses as delineated in (2).

As can be seen, parallel file systems provide much richer semantics and are more flexible in terms of supported file access patterns and data sharing than distributed file systems. Their algorithms are carefully designed to avoid communication and synchronization hotspots while maintaining high-bandwidth access to file data whenever possible, providing stronger guarantees of data integrity, and supporting the necessary level of fault resilience and availability. Of the top 10 machines on the Top 500 list, Cori at National Energy Research Scientific Computing Center, Mira at Argonne National Laboratory, and Piz Daint at Centro Svizzero di Calcolo Scientifico (Switzerland) use GPFS to manage respectively 30, 27, and 5.8 PB of storage.

18.5 LUSTRE FILE SYSTEM

Lustre is a parallel distributed file system originally released in 2003. Its name is derived from "Linux" and "clusters", indicating the intended target platforms for its deployment. Its development was initially carried out under the Department of Energy Accelerated Strategic Computing Initiative (ASCI) Path Forward [12]. Corporate ownership of the project and its code base changed hands several times and has included Sun Microsystems, Oracle, Whamcloud, and, since 2012, Intel.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 555

Lustre provides a POSIX-compliant file system interface with atomic semantic support for most operations, thus avoiding data and metadata inconsistencies. It design is highly scalable, making it a preferred file system for HPC by supporting multiple tens of thousands clients, petabytes of storage, and I/O bandwidths reaching multiple hundreds of GB/s. Deployment of multiple clusters is simplified with Lustre, as it permits aggregation of both capacity and performance of multiple storage subsystems. The storage space and I/O throughput can be also dynamically increased by providing additional storage servers as needed. Lustre takes advantage of high performance networking infrastructure, such as low-latency communication and remote direct memory access (RDMA) over InfiniBand with OpenFabrics Enterprise Distribution (OFED) [13]. Lustre software enables the bridging of multiple RDMA networks and provides integrated network diagnostics. The file system supports high availability with multiple failover modes using shared storage partitions and interfacing with different high-availability managers. This implements automatic failovers with no single point of failure, as well as transparent application recovery. The chances of file system corruption are minimized through a multiple-mount protection feature. Particularly noteworthy is the online distributed file system check (LFSCK) that is capable of operating while the file system is in use to restore data consistency after a major file system error is detected. Security of operation is enforced by permitting TCP connections only on privileged ports and application of ACLs and extended attributes based on POSIX ACLs with custom additions, such as *root squash* (reduction of effective access rights for the remote superuser). Lustre uses a distributed lock manager (LDLM) to permit file locking with byte granularity as well as fine-grain metadata locks to permit concurrent operation of multiple clients on the same files and directories. File striping across physical storage devices permits the user to specify the layout parameters, which may be flexibly arranged at the level of a whole file system, a single directory, or individual files to match the needs of specific applications. Lustre is highly interoperable; it supports a dedicated MPI-IO abstract-device interface for I/O layer to provide optimized parallel I/O to message-passing interface (MPI) applications and permits exports of its files through commonly used distributed file system interfaces such as NFS and CIFS, enabling access to its files from non-Unix hosts. The Lustre code base compiles and runs on a variety of hardware platforms, including machines of different endianness and native data sizes, and transparently interfaces with older revisions of file system software. Lustre software is open sourced under the GNU public license 2.0 license; its current major revision is v2.8. Many of these features account for the popularity of Lustre deployment in HPC systems: as of November 2016 half the 10 fastest supercomputers on the Top 500 list (Tianhe 2, Titan, Sequoia, Oakforest-PACS, and Trinity) integrated Lustre as the main storage management layer.

A schematic view of Lustre architecture is shown in Fig. 18.4. The primary functional components of a Lustre system are as follows:

- *Management server* (MGS) is responsible for storing, managing, and supplying the configuration information to other Lustre components. It interacts with all targets (configuration providers) and clients (configuration accessors) in the system. While MGS typically works using a dedicated set of storage devices for independent operation, the storage could also share the physical devices present in the metadata server pool.
- *Management target* (MGT) provides storage space for the management server. Its space requirements rarely exceed 100 MB even in large-scale Lustre installations. While the performance of the underlying storage is not critical for the operation of the system

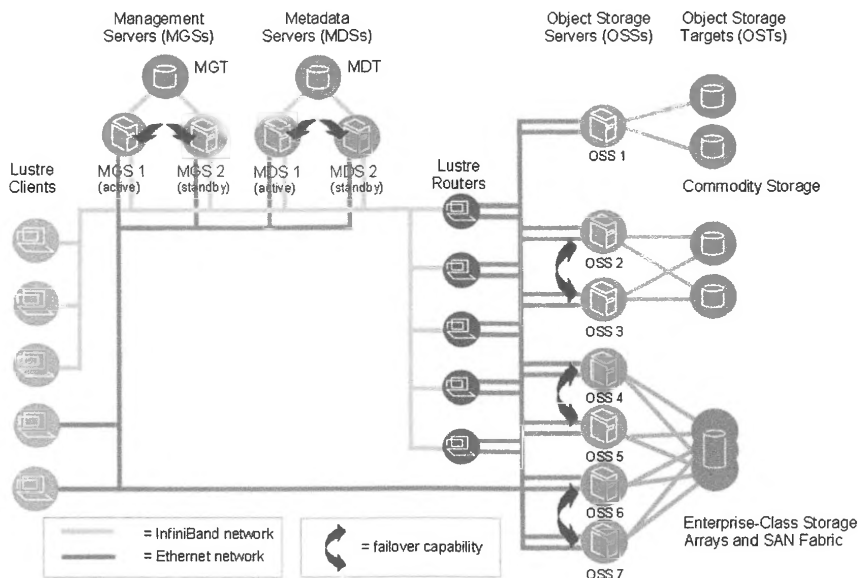


FIGURE 18.4

Layout of typical Lustre deployment at scale.

Image via Lustre.org

(seeks and writes of small amounts of data), its reliability is paramount. MGT may leverage redundant storage structures such as RAID1 to provide it. Multiple MDTs per system are supported.

- *Metadata server* (MDS) that is responsible for management of the names and directory contents. The namespace in Lustre may be distributed across multiple MDSs. Each MDS also handles network requests for one or more MDTs. MDS failovers are supported: a standby MDS assumes the functions of a failed active MDS.
- *Metadata target* (MDT) that stores various metadata, including directories, file names, permissions, and file layout information on physical storage associated with an MDS. There is nominally one MDT per file system, although recent revisions support multiple MDTs under the distributed namespace environment (DNE). The primary MDT comprises the root of the file system, while the additional MDSs with their own attached MDTs may hold various subdirectories. It is also possible to distribute the contents of a single directory across multiple MDT nodes, thus creating a *striped directory*. MDT storage usually accounts for 1%–2% of the total file system capacity.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 557

- *Object storage server* (OSS) that services file data I/O requests and other network requests for one or more object storage target (OST). A common Lustre configuration involves an MDT on a dedicated hardware node, two or more OSTs on every OSS node, and an I/O client on every compute node of a system. The ratio of OSTs to OSSs typically varies between two and eight.
- *Object storage target* (OST) that manages physical storage for user file contents. The file data are contained in one or more objects, each of which is under control of a specific separate OST. The number of objects a file is divided into is configurable by the user. Single OST capacity is limited to 128 TB (256 TB on ZFS, an advanced file system originally developed at Sun Microsystems); the total file system capacity is the sum of capacities of all OSTs.
- *Clients* that execute the applications generating the I/O data. They may include conventional compute nodes, but also loosely associated desktops, workstations, or visualization servers that are permitted to mount the file system.
- *Lustre Networking* (LNET) that provides the communication infrastructure for the whole system. Its main features include concurrent access to and support of many common network types (IB/OFED, TCP variants, including GigE, 10GigE, and IPoIB, Cray Seastar, Myrinet MX, Rapid Array, and Quadrics Elan), RDMA (if available), routing between individual network segments, high availability, and recovery from network errors. LNET strives to achieve end-to-end communication bandwidth nearing the available peak bandwidth. Its software includes the higher-level code module and the underlying network driver (LND). The LNET layer is connectionless and asynchronous, leaving the verification of data transmission status to the connection-oriented LND. Bonding of multiple network interfaces for increased bandwidth is also supported.

The high-level organization of a file in Lustre is depicted in Fig. 18.5. The files are referred to by 128-bit file identifiers (FIDs) that consist of a unique 64-bit sequence number, a 32-bit object ID (OID), and a 32-bit version number. FID identifies an object in MDT whose extended attributes encode the layout information: one or more pointers to OST objects that contain the file data. Since the objects must be stored on different OSTs, the data are striped in a round-robin fashion across all OSTs (obviously, no striping is applied if only one OST is associated with the file). The number of stripes, stripe size, and target OSTs are user configurable. The default stripe count is one and the default stripe size is 1 MB. There may be up to 2000 objects per file. Since the client performing data I/O operations on a file must first fetch the layout extended attribute data from the MDT object identified by FID, further data transfers can be arranged directly between the client node and the related OSS nodes storing the file data.

Efficient synchronization of file operations in parallel file systems is a key factor in achieving a good level of performance. Lustre resources are associated with locks that may be local or global. LDLM is based on a locking algorithm utilized by VAX DLM [14]. To give the reader an idea of the complexity, a brief overview of the involved data structures and algorithms is presented below.

LDLM locks may exist in one of six modes.

- *Exclusive* mode requested by MDS before a new file is created.
- *Protective Write* mode issued by the OST to the client requesting a write lock.
- *Protective Read* mode granted by the OST to clients that need to read or execute files.
- *Concurrent Write* mode issued by the MDS to clients requesting write lock when opening a file.
- *Concurrent Read* mode associated with intermediate path traversal during path lookups and effected by the related MDS.
- *Null* mode.

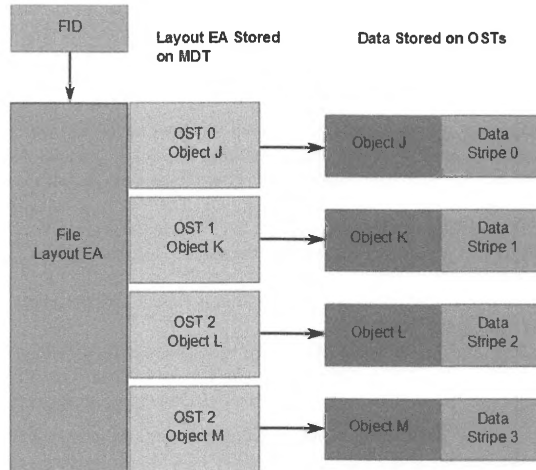


FIGURE 18.5

Lustre file layout.

In addition, Lustre defines four types of locks:

- *extent lock* for OST data protection
- *flock* required to support user space requests for file locking
- *inode bit lock* to protect metadata attributes
- *plain lock*, usually unused.

Lock management supports three types of callback functions. *Blocking callback* is invoked when a client requests a lock conflicting with the current one, giving the client an opportunity to renounce the lock or the lock is forcibly revoked. *Completion callback* is called when a requested lock is granted or a lock is converted to a different mode. Finally, a *glimpse callback* is used to provide certain information about file without releasing the held lock. LDLM also uses the concepts of *namespace* and *intent*. Each service in Lustre, such as OST, MDS, and MGS, is associated with a namespace. In turn, the intent is a small amount of data indicating that special processing must be invoked during the lock processing operation. Each namespace has potentially several different intent handlers to support that. The two fundamental operations, lock request and release, are controlled by precisely defined algorithms. Thus to obtain a lock, the following actions must occur.

1. A client locking service determines if the lock belongs to a local namespace. If it is local, the algorithm advances to (7).

2. A lock enqueue RPC is sent to LDLM on the appropriate server. An initial ungranted lock is created, with some fields initialized from data supplied by the request.
3. The enqueue step inspects if there is an intent set on the lock. If not, it invokes the policy function associated with the lock type. The policy function determines whether the lock may be granted or not. If the intent on the lock is set, the algorithm proceeds to (6).
4. The server then checks if there are any conflicts with already granted and waiting locks for the resource specified by the request. If no conflict is found, the lock is granted. A completion callback is invoked and the lock is acquired. Otherwise, the processing continues in (5).
5. A blocking callback is invoked for every conflicting lock. The lock may be held at the client, in which case an RPC request is emitted; otherwise, a flag is set at the server. After all the locks are scanned, the processed lock request is entered on the waiting list and the lock is returned to the client with its status set to "blocked".
6. After the lock intent is set, an appropriate intent handler is called. LDLM returns the result of the call without further interpretation.
7. Local locks are created and then enqueued to check if they can be granted as described above. This process continues without any RPCs. If the lock can be granted or errors are detected, the control returns immediately with the lock status correctly marked. Otherwise, the lock request is blocked.

Typically, locks in Lustre are held indefinitely. Lock release is initiated when another process requests a conflicting lock, a blocking callback is issued by LDLM, or a blocking callback is invoked on the client node. The lock cancellation proceeds as follows.

1. If the sum of active readers and writers is nonzero, it means that another process on the same client is using the lock and no action is taken. The lock owner(s) will eventually release it.
2. There are no readers or writers. A blocking callback is invoked with a flag indicating lock revocation.
3. If the lock is not in the local namespace, an RPC call is sent to the client containing a cancellation request. Otherwise, local cancellation is performed that takes the lock off all the lists.
4. All waiting locks on the resource are reevaluated.
5. If any of the waiting locks can be granted, they are moved to the granted lock list and a completion callback is invoked.

One of Lustre's strengths is fault management, which can be applied to most of its functional components. Two basic failover modes are available: active/passive and active/active. In the first configuration the active server processes client requests and provides resources, while the passive server stays idle. In case of active node failure, the passive server becomes active and takes over. The second scenario involves multiple active servers, each providing a subset of resources. If one fails, the remaining ones take over the failed node's resources. A variation of these schemes is also used to provide better utilization of system resources. For example, an idle server in active/passive configuration for one Lustre cluster may at the same time be the active server for another file system.

An overview of various operational parameters of the Lustre file system is presented in Table 18.2. Since the underlying file system can be selected by the system administrator as either *ldiskfs* (a modified and patched revision of the Linux *ext4* journaling file system) or ZFS, some of the absolute

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 560

Table 18.2 Select Operational Parameters of Lustre

Parameter	Design Target	Production Tested
Maximum file size	31.25 PB (ldiskfs) 16 TB (32-bit ldiskfs) 8 EB (ZFS)	Multiple TB
Maximum file count	32 billion (ldiskfs) 256 trillion (ZFS)	2 billion
Maximum storage space	512 PB (ldiskfs) 1 EB (ZFS)	55 PB
Number of clients	≤131,072	50,000+
Single-client I/O performance	90% network bandwidth	2 GB/s data I/O 1000 metadata ops/s
Aggregate-client I/O performance	10 TB/s	2.5 TB/s
OSS count	1000 OSSs, up to 4000 OSTs	450 OSSs with 1000 4 TB OSTs 192 OSSs with 1344 8 TB OSTs 768 OSSs with 768 72 TB OSTs
Single OSS performance	10 GB/s	6+ GB/s
Aggregate OSS performance	10 TB/s	2.5 TB/s
MDS count	≤256 MDTs, ≤256 MDSs	1 primary and 1 backup
MDS performance	50,000 create ops/s 200,000 stat ops/s	15,000 create ops/s 50,000 stat ops/s

Excerpted from Intel Corp., Lustre Software Release 2.x Operations Manual [Online]. Available: http://doc.lustre.org/lustre_manual.pdf.

limits listed depend on the file system type used. As Lustre continues to be deployed in installations of increasing scales and capacities, some of the listed configurations tested in production may be out of date by the time of publication.

18.6 SUMMARY AND OUTCOMES OF CHAPTER 18

- File systems provide an abstraction necessary to manage the information kept on mass-storage devices. They organize the information in a hierarchical layout, provide human-accessible namespace to identify individual stored entities uniquely, maintain attributes describing access permissions and various properties of individual entries, verify the consistency of stored information, provide fault-recovery mechanisms, and expose the user interface for access. File systems achieve these by defining and manipulating additional *metadata* that describes the layout and various properties of the stored raw data.
- Distributed file systems are file systems that are capable of handling I/O requests issued by multiple clients over the network. To manage the demands of scaling, they frequently span multiple server nodes while providing a "single view" access to the stored data and related namespace.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 561

- Parallel file systems are distributed file systems that are specifically optimized to support concurrent file access efficiently from parallel applications. In particular, they implement synchronization mechanisms that permit the distributed application to operate on different sections of the same file or enable strided access for individual clients accessing the same file while preserving the consistency of data and metadata for multiple accessors.
- The POSIX standard defines a local file access interface in Unix environments. Two modes of access are commonly supported by the runtime library: one based on system calls and another on buffered file I/O (streams).
- NFS is one of the most frequently deployed distributed file systems in small and medium cluster environments. It permits the use of the POSIX interface and implements session semantics in which the clients most efficiently operate on disjoint files with updates propagated at the end session (file close). The available features and performance strongly depend on the installed NFS code revision and configuration.
- GPFS is a high performance proprietary parallel file system designed for scalability and high-bandwidth concurrent file access. It implements token-based locking of arbitrary shared file sections and synchronization techniques that identify concurrent file access conflicts and guarantee consistency of the affected data and metadata.
- Lustre is a high performance open-source parallel file system supporting multiple network types and host architectures. Due to its good performance, permissive licensing, and extensive list of features (dynamic expandability, multiple network support, RDMA, failover for multiple components, sophisticated distributed file lock management, POSIX and MPI-IO interfaces, NFS and CIFS export support, and many others), it is frequently used in large-scale cluster installations.

18.7 QUESTIONS AND PROBLEMS

1. Summarize the main challenges of creating efficient persistent data storage for an HPC system. How may they be solved?
2. What are the differences between system-call-based and streaming I/O interfaces in POSIX? What are their implications for file access performance?
3. Write a program that saves an array of 1000 double-precision floating-point numbers to a file using in-memory layout and an array of 1000 structures consisting of one character and one double-precision number to another file. Do the sizes of the generated files match the estimated values based on the sizes of the involved elementary data types multiplied by array size? If not, what is the reason for the discrepancy? Can the inefficiency (if any) be eliminated?
4. A computational scientist attempts to debug his stubbornly crashing MPI application. Due to a complicated sequence of events leading to the crash, he gets an idea to use a shared log file located on an NFS partition to store the information about the event occurrences on every node. When analyzing the file he begins to suspect that not all captured data were actually written to file. What may be the reason for that? How would you improve the reliability of logging the precrash data?

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 562

5. Consider the following code that prints array elements to a file and reads them back.

```
1 #include <stdio.h>
2
3 #define SIZE 512
4 #define FILENAME "myfile"
5
6 int main() {
7   double data[SIZE], iodata[SIZE];
8   for (int i = 0; i < SIZE; i++) data[i] = i+1/(double)(i+1);
9
10  FILE *f = fopen(FILENAME, "w");
11  for (int i = 0; i < SIZE; i++) fprintf(f, "%lf\n", data[i]);
12  fclose(f);
13
14  f = fopen(FILENAME, "r");
15  for (int i = 0; i < SIZE; i++) {
16    fscanf(f, "%lf", &iodata[i]);
17    if (data[i] != iodata[i])
18      printf("ERROR: item %d should be %lf, got %lf\n", i, data[i], iodata[i]);
19  }
20  fclose(f);
21  return 0;
22 }
```

a. Is running the code going to produce any error messages? Why? Verify your answer by compiling and executing the program.
b. How would you fix the encountered problem(s)?
c. Based on this experience, would you recommend saving floating-point data as text? Justify your answer.

6. Contrast distributed and parallel file systems. Which solutions provided by the latter improve the efficiency of concurrent accesses to shared files?

REFERENCES

[1] IEEE and The Open Group, The Open Group Base Specifications Issue 7, IEEE Standard 1003.1−2008, 2016 Edition, [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799.

[2] ISO/IEC 9899:201x C Language Standard Draft, April 12, 2011 [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[3] IETF Network Working Group, RFC 5531: RPC: Remote Procedure Call Protocol Specification Version 2, May, 2009 [Online]. Available: https://tools.ietf.org/html/rfc5531.

[4] IETF Network Working Group, RFC 4506: XDR: External Data Representation Standard, May, 2006 [Online]. Available: https://tools.ietf.org/html/rfc4506.

[5] R.H. Arpaci-Dusseau, A.C. Arpaci-Dusseau, The Andrew File System (AFS), in: Operating Systems: Three Easy Pieces, Arpac-Dusseaui Books, 2014.
[6] Microsoft TechNet Library, Common Internet File System, Microsoft, [Online]. Available: https://technet.microsoft.com/en-us/library/cc939973.aspx.
[7] Kerberos: The Network Authentication Protocol, Massachusets Institute of Technology, November 16, 2016 [Online]. Available: http://web.mit.edu/kerberos/.
[8] IETF Network Working Group, RFC 2847: LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM, June, 2000 [Online]. Available: https://tools.ietf.org/html/rfc2847.
[9] IETF, RFC 5661: Network File System (NFS) Version 4 Minor Version 1 Protocol, January, 2010 [Online]. Available: https://tools.ietf.org/html/rfc5661#page-277.
[10] R.L. Haskin, F.B. Schmuck, The Tiger Shark File Syetem, in: Compcon '96: Technologies for the Information Superhighway, 1996.
[11] P.F. Corbett, D.G. Feitelson, The Vesta parallel file system, ACM Transactions on Computer Systems 14 (3) (1996) 225–264.
[12] G. Grider, The ASCI/DOD Scalable I/O History and Strategy, May, 2004 [Online]. Available: https://www.dtc.umn.edu/resources/grider1.pdf.
[13] OFED Overview, OpenFabrics Alliance, [Online]. Available: https://www.openfabrics.org/index.php/openfabrics-software.html.
[14] N.P. Kronenberg, H.M. Levy, W.D. Strecker, VAXclusters: a closely-coupled distributed system, ACM Transactions on Computer Systems 4 (2) (1986) 130–146.
[15] Intel Corp., Lustre Software Release 2.x Operations Manual, [Online]. Available: http://doc.lustre.org/lustre_manual.pdf.

CHAPTER

MAPREDUCE

19

CHAPTER OUTLINE

19.1 Introduction 579
19.2 Map and Reduce 579
19.2.1 Word Count 581
19.2.2 Shared Neighbors 581
19.2.3 K-Means Clustering 582
19.3 Distributed Computation 584
19.4 Hadoop 585
19.5 Summary and Outcomes of Chapter 19 588
19.6 Exercises 589
References 589

19.1 INTRODUCTION

MapReduce is a simple programming model for enabling distributed computations, including data processing on very large input datasets, in a highly scalable and fault-tolerant way. While the concept of MapReduce was motivated initially by functional programming languages like *LISP* with its *map* and *reduce* primitives, it is also closely related to the message-passing interface (MPI) concepts of *scatter* and *reduce* for distributed-memory architectures. However, unlike in MPI programming, the details of the underlying parallelization in MapReduce are hidden from the programmer, making it easier to use. MapReduce algorithms have been shown to scale from single servers all the way to hundreds of thousands of cores while at the same time delivering transparent fault tolerance to the end user. MapReduce was developed by Google [2], and the programming model has since been adopted by many software frameworks, libraries, and end users. Apache's open-source Hadoop framework [1] is one of several libraries which support MapReduce, and is used for the examples in this chapter.

19.2 MAP AND REDUCE

A *map* is a functional that executes a supplied function on all members of an input list. Because the map function only requires the input data member to execute, it can be run in parallel, providing a massive potential speed-up. The map function itself returns a set of two linked data items: a key for lookup and a value. The key can either be the output from the function or the input data element itself.

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00019-8
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 565

For example, suppose the map function counts the number of characters of an input word, returning as a key the word length and returning as a value the input word. So if the word "computing" is supplied to the map function, it would return the key—value pair of "9:computing" where the key is "9", the length of the word "computing", and the corresponding value to the key is the input data element word, "computing". The keys from the output are then grouped by key after executing the map function on each data element. For example, if the same map function were executed on each word in the sentence "This is a book about high performance computing", the result of the map portion in MapReduce would be the groupings shown in Table 19.1.

The results of the map function and associated groupings are then passed to the *reduce* function. The reduce function takes as an argument a key and all values associated with that key. Like the map function, the reduce function can also be executed independently on each key and grouping of values, thereby enabling embarrassingly parallel execution. For example, suppose a crossword-puzzle designer would like to know the number of words with a length of four characters that occur in a large-input dataset. The reduce function in this case would simply count the number of grouped values associated with each key. Using the previous map function example, the output from the reduction function in this case would be as shown in Table 19.2.

In this example, there are three words with a length of four characters and the rest are all of length one.

From the user's perspective, some of the principal strengths of the MapReduce programming model are that the parallelization and fault-tolerance details of the MapReduce implementation are hidden from the user and only the map and reduce functions need to be supplied. Map and reduce functions themselves vary widely in complexity. The following subsections give some additional examples of map and reduce functions.

Table 19.1 Sample MapReduce Map Function Groupings

Key	Grouped Values
1	"a"
2	"is"
4	"this", "book", "high"
9	"computing"
11	"performance"

Table 19.2 Sample MapReduce Reduce Function Groupings

Key	Output From Reduce Function
1	1
2	1
4	3
9	1
11	1

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 566

19.2.1 WORD COUNT

Counting the number of times each word has been used in a body of text is the canonical didactic example for MapReduce. The map function returns as a key a single word and the associated value with the key is unity. For example, the result for this map function on the famous text from Shakespeare's *Hamlet*, "To be or not to be—that is the question", is as shown in Table 19.3.

Because the words "to" and "be" occur twice, the value of 1 is added to the grouped values twice (once per occurrence).

The reduce function simply sums up the grouped values for each key, as illustrated in Table 19.4.

Running this map and reduce function on the entire text of Shakespeare's *Hamlet* gives the word counts for some common words, as shown in Table 19.5.

19.2.2 SHARED NEIGHBORS

Finding shared neighbors in graph applications provides another good example of MapReduce functionality. A sample graph is shown in Fig. 19.1, where multiple vertices share the same neighbors. For example, in this graph vertex "0" shares a common neighbor with vertex "2"; this common neighbor is vertex "1". MapReduce can be used to find those shared neighbors.

Table 19.3 Sample MapReduce Word Counts

Key	Grouped Values
"to"	1, 1
"be"	1, 1
"or"	1
"not"	1
"that"	1
"is"	1
"the"	1
"question"	1

Table 19.4 Sample MapReduce Reduce Function Output

Key	Output From Reduce Function
"to"	2
"be"	2
"or"	1
"not"	1
"that"	1
"is"	1
"the"	1
"question"	1

Table 19.5 Sample MapReduce Word Counts

Key	Output From Reduce Function
"but"	269
"as"	222
"be"	210
"England"	21
"Norway"	13

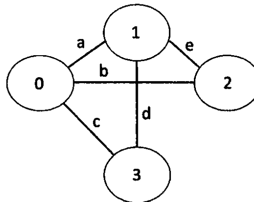


FIGURE 19.1

A small graph where the vertices share multiple neighbors. Vertices are listed as numbers (0–3) and edges are lower-case letters (a–e).

In this case, the map function returns each edge of a vertex as a key. The value for each key is the list of all the neighboring vertices to that vertex (Table 19.6).

This gives the group values shown in Table 19.7.

The reduce function returns the intersection of each key's grouped values, thereby revealing shared neighbors for each edge (Table 19.8).

This simple map and reduce operation reveals the shared neighbors between any two connected vertices. For example, vertices connect by edge "a" in Fig. 19.1 (vertices 0 and 1) also share two of the same neighbors, vertices 2 and 3.

19.2.3 K-MEANS CLUSTERING

K-means clustering partitions a data space into *k* clusters, each with a mean value. Each individual in the cluster is placed in the cluster closest to the cluster's mean value. K-means clustering is frequently

Table 19.6 Values of Vertices in Fig. 19.1

Vertex 0		Vertex 1		Vertex 2		Vertex 3	
Key	Values	Key	Values	Key	Values	Key	Values
a	1,2,3	a	0,2,3	e	0,1	c	0,1
b	1,2,3	d	0,2,3	b	0,1	d	0,1
c	1,2,3	e	0,2,3				

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 568

Table 19.7 Grouped Values of Vertices in Fig. 19.1

Key	Grouped Values
a	(1,2,3), (0,2,3)
b	(1,2,3), (0,1)
c	(1,2,3), (0,1)
d	(0,2,3), (0,1)
e	(0,2,3), (0,1)

Table 19.8 Shared Neighbors in Fig. 19.1 Revealed in MapReduce

Key	Output From Reduce Function
a	2,3
b	1
c	1
d	0
e	0

used in data analysis, and a simple example with five x and y value pairs to be placed into two clusters using the Euclidean distance function is given in Table 19.9.

To begin the clustering, two initial cluster points are supplied: (0,0) and (1,1). Using the Euclidean distance measure, $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, each individual is assigned to the cluster nearest to the (x,y) pair, as summarized in Table 19.10.

The initial cluster points have moved from (0,0) to (0.1,0.3) and from (1,1) to (0.8,0.85). This same process can be repeated until the cluster mean values stop changing or a maximum number of iterations is reached.

In a MapReduce programming model, for a given (x,y) value pair the mapper iterates over each cluster's mean value and finds the cluster with the nearest distance to the (x,y) pair. It returns as a key the cluster and as a value the (x,y) pair (Table 19.11).

Table 19.9 Example of K-Means Clustering

Individual	(x,y) Pair
a	(0.1,0.3)
b	(1.1,0.4)
c	(0.8,0.7)
d	(1.2,1.2)
e	(0.1,1.1)

Table 19.10 Assignment of Cluster Members

Cluster 1		Cluster 2	
Members	**Mean x,y Value**	**Members**	**Mean x,y Value**
a	(0.1,0.3)	b,c,d,e	(0.8,0.85)

Table 19.11 MapReduce K-Means Clustering

Key	Grouped Values
1	(0.1,0.3)
2	(1.1,0.4), (0.8,0.7), (1.2,1.2), (0.1,1.1)

The reducer receives a list of (x,y) value pairs for each cluster and computes the new cluster mean value (Table 19.12).

This MapReduce operation can be performed iteratively until no more updates occur or a maximum number of iterations is reached.

19.3 DISTRIBUTED COMPUTATION

Distributed processing in MapReduce may be summarized in three phases: a map phase, a shuffle phase, and a reduce phase. These phases can be overlapped to some degree to improve efficiency. The map step applies the map function to data local to the processor. Input data for MapReduce is frequently stored in a distributed file system where data blocks are already shared between different linked storage devices, with some redundancy for fault tolerance. The map function does not operate on redundant copies. The shuffle step relocates the map output data based on the output key from the map function so that map output is grouped by output key. The reduce step applies the reduce function to the output data from the map function.

The map functions, like the reduce functions, can be executed concurrently giving a significant potential for speedup. However, efficient distributed MapReduce execution generally requires minimizing the movement of data. For example, it is more efficient for the nodes performing map functions to execute the map on blocks local to the node. Similarly, in the shuffle and reduce phases, the movement of data can be reduced by executing reduce functions on nodes where the map output data already resides.

Table 19.12 MapReduce Cluster Mean Values

Key	Output From Reduce Function
1	(0.1,0.3)
2	(0.8,0.85)

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 570

19.4 HADOOP

The Hadoop project by Apache [1] is an open-source implementation of the MapReduce programming model. It provides a distributed file system, job scheduling and resource management tools, including YARN (Yet Another Resource Negotiator), and MapReduce programming support. Historically, MapReduce applications in Hadoop are programmed using Java, although support for C++, Python, and a few other languages is also available.

The Hadoop distributed file system (HDFS) enables distributed file access across many linked storage devices in an easy way. It was motivated by the Google file system, which was instrumental in the original MapReduce programming model development [3]. Data in the distributed Hadoop file system is broken into blocks and distributed across the linked storage devices on the system. Blocks are generally replicated at least once to guard against storage or machine failures depending upon the fault-tolerance properties used when configuring Hadoop. File system commands are run on the Hadoop distributed file system using the *hdfs dfs* command. A summary of the most commonly used file system commands for hdfs dfs are listed in Table 19.13.

As an example, a text file of Shakespeare's *Hamlet* (*hamlet.txt*) stored in the local file system can be placed in HDFS as follows:

```
hdfs dfs -put hamlet.txt /hamlet
```

The contents of hdfs can be queried using the "ls" command on the "/" directory:

```
hdfs dfs -ls /
```

This file can now be used in conjunction with a MapReduce operation inside Hadoop.

As an example MapReduce application, the word count MapReduce from Section 19.2.1 is implemented in Hadoop using the Java programming language in Fig. 19.2. The mapper function, which returns a single word for the key and unity for a value, is illustrated in lines 69–82. The reducer, which returns the single word as a key and the sum of the grouped values provided from the mapper as a value, is illustrated in lines 87–98.

Table 19.13 Select Hadoop Distributed File System Commands

Select Hadoop Distributed File System Commands	Description
hdfs dfs –cat <filename>	Copies the specified filename to stdout.
hdfs dfs –ls	HDFS equivalent of Linux "ls" command.
hdfs dfs –mkdir <directory>	HDFS equivalent of Linux "mkdir" command
hdfs dfs –put <local files>... <destination>	Copy the source files to the destination path in HDFS
hdfs dfs –get <src> <destination>	Copy the source file to the local file system destination
hdfs dfs –rm <filenames>	Delete the specified files; only deletes files
hdfs dfs –rmr <directory name>	Delete the specified directory and all content

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 571

```
0001 import java.io.IOException;
0002
0003 // need StringTokenizer for space delimited input
0004 import java.util.StringTokenizer;
0005
0006 // Needed for filesystem path (lines 49-50)
0007 import org.apache.hadoop.fs.Path;
0008
0009 // Needed for providing job configuration
0010 import org.apache.hadoop.conf.Configuration;
0011
0012 // Needed for Hadoop data wrappers like Text and IntWritable
0013 import org.apache.hadoop.io.*;
0014
0015 // MapReduce
0016 import org.apache.hadoop.mapreduce.Mapper;
0017 import org.apache.hadoop.mapreduce.Reducer;
0018 import org.apache.hadoop.mapreduce.Job;
0019 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
0020 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
0021
0022 public class HamletCounter {
0023
0024  public static void main(String[] args) throws Exception {
0025
0026    // check that two arguments are supplied:  the input data and output
location
0027    if (args.length != 2)
0028    {
0029      System.out.println("Takes two arguments: <data in> <result>");
0030      System.exit(0);
0031    }
0032
0033    // Set up the job configuration
0034    Configuration config = new Configuration();
0035
0036    // Give a name to the job:  "Counting Hamlet"
0037    Job job = new Job(config, "Counting Hamlet");
0038
0039    // Use Hadoop data types:  in both the mapper and the reducer
0040    //   the key is a string and the value is an int.  The Hadoop
0041    //   equivalents to string and int are Text and IntWritable, respectively.
0042    job.setOutputKeyClass(Text.class);
0043    job.setOutputValueClass(IntWritable.class);
0044
0045    // Give the job the names of the map and reduce classes
0046    job.setReducerClass(reducing.class);
0047    job.setMapperClass(mapping.class);
0048
0049    FileInputFormat.addInputPath(job, new Path(args[0]));
0050    FileOutputFormat.setOutputPath(job, new Path(args[1]));
0051
0052    // start
0053    job.waitForCompletion(true);
0054  }
```

FIGURE 19.2

Example code, HamletCounter.java, using Hadoop. The mapper is in lines 69—82 and returns each word as a key and unity as a value. The reducer is in lines 87—98 and returns the word as a key and the sum of the list of values it receives from the reducer.

```
0055
0056  public static class mapping extends Mapper<LongWritable, Text, Text,
IntWritable> {
0057
0058    // IntWritable is the Hadoop version of an integer optimized for Hadoop
0059    private final static IntWritable unity = new IntWritable(1);
0060
0061    // Usage of Hadoop data wrappers was set in lines 42-43
0062    // Use Text instead of Java's String class for output
0063    private Text single_word = new Text();
0064
0065    // The Hadoop MapReduce framework calls map(Object, Object, Context)
0066    // The key is a LongWritable -- Hadoop's version of long
0067    // The value is a Text -- Hadoop's version of String
0068    // Context objects are used for writing output pairs from mappers and
reducers
0069    public void map(LongWritable key, Text val, Context output)
0070                              throws IOException, InterruptedException {
0071      // Converting the input line of text from Hadoop's Text to a String
0072      String text_line = val.toString();
0073
0074      // Split the line into space delimited
0075      StringTokenizer space_delimited = new StringTokenizer(text_line);
0076      while (space_delimited.hasMoreTokens()) {
0077        single_word.set(space_delimited.nextToken());
0078
0079        // Here we write a single word as the key and give it a value of unity
0080        output.write(single_word, unity);
0081      }
0082    }
0083  }
0084
0085  public static class reducing extends Reducer<Text, IntWritable, Text,
IntWritable> {
0086
0087  public void reduce(Text key, Iterable<IntWritable> grouped_values, Context
output)
0088                                         throws IOException,
InterruptedException {
0089    int sum_of_times_word_is_used = 0;
0090    for (IntWritable single_value : grouped_values) {
0091      sum_of_times_word_is_used += single_value.get();
0092    }
0093
0094    // Hadoop data wrappers are set to be used in lines 42-43 so the output
can't be an int;
0095    //  make it an IntWritable
0096    IntWritable total_times_word_used = new
IntWritable(sum_of_times_word_is_used);
0097    output.write(key, total_times_word_used);
0098    }
0099  }
0100 }
```

FIGURE 19.2 Cont'd

The application requires two arguments: the input data file placed in HDFS and the output directory where results from the reducer will be written. The code is compiled using *javac* and the compiled classes are placed in a subdirectory called *build*:

```
mkdir build
javac -cp $(hadoop classpath) -d build HamletCounter.java
```

A Java archive file, *hamletcount.jar*, is then created using the compiled class files in the build directory in preparation for execution by Hadoop:

```
jar -cvf hamletcount.jar -C build
```

Hadoop then executes the Java archive file as follows:

```
hadoop jar hamletcount.jar HamletCounter /hamlet /hamlet_result
```

where */hamlet* and */hamlet_result* are the input and output arguments required by the program. The */hamlet* text was already added to HDFS and the output from the MapReduce execution will be written to the */hamlet_result* directory. This data can be retrieved from the distributed file system to the local file system using *hdfs dfs −get* as follows:

```
hdfs dfs -get /hamlet_result
```

This will copy the entire directory of */hamlet_result* to the local file system with the results of the word count for *Hamlet*.

19.5 SUMMARY AND OUTCOMES OF CHAPTER 19

- MapReduce is a simple programming model for enabling distributed computations, including data processing on very-large-input datasets in a highly scalable and fault-tolerant way.
- The details of the underlying parallelization in MapReduce are hidden from the programmer, thereby making it easier to use.
- A *map* is a functional that executes a supplied function on all members of an input list.
- The results of the map function and associated groupings are passed to the *reduce* function.
- The map functions, like the reduce functions, can be executed concurrently giving a significant potential for speed-up.
- Distributed processing in MapReduce may be summarized in three phases: a map phase, a shuffle phase, and a reduce phase.

- Efficient distributed MapReduce execution generally requires minimizing the movement of data.
- The Hadoop project provides an open-source implementation of the MapReduce programming model.
- Hadoop provides a distributed file system, job scheduling and resource management tools, and MapReduce programming support

19.6 EXERCISES

1. By either using the word counter map and reduce functions from Fig. 19.2 or creating your own, discover how many times the word Denmark is used in Shakespeare's *Hamlet*. Then find William Shakespeare's top 10 most used words by applying your word counter tool to all the works of Shakespeare.
2. Implement the map and reduce functions of the shared neighbor finder in the graph problems presented in Section 19.2.2. Apply this map—reduce operation to the IMDb movie database [4] to find the common costar links between 10 famous actors or actresses.
3. Implement the map and reduce functions of the K-means clustering algorithm presented in Section 19.2.3. Generate a random set of x and y points and execute K-means clustering on this set. Plot the time to solution as a function of set size.
4. Using a full Wikipedia dump [5] as input, find the 20 most common words in the 10 most widely spoken languages [6] (Mandarin, Spanish, English, Hindi, Arabic, Portugese, Bengali, Russian, Japanese, and Punjabi).

REFERENCES

[1] Apache, Apache Hadoop. [Online] http://hadoop.apache.org/.

[2] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, s.n., San Francisco, 2004.

[3] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google file system, in: 19th ACM Symposium on Operating Systems Principles, ACM, Lake George, NY, 2003.

[4] IMDb, Plain Text Data Files for IMDb FTP Site. [Online] ftp://ftp.fu-berlin.de/pub/misc/movies/database/.

[5] Wikipedia, Wikipedia Downloads. [Online] https://dumps.wikimedia.org/.

[6] List of Languages by Number of Native Speakers, Wikipedia. [Online] https://en.wikipedia.org/wiki/List_of_languages_by_number_of_native_speakers.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 575

CHAPTER

CHECKPOINTING

20

CHAPTER OUTLINE

20.1 Introduction 591
20.2 System-Level Checkpointing 592
20.3 Application-Level Checkpointing 598
20.4 Summary and Outcomes of Chapter 20 602
20.5 Exercises 602
References 603

20.1 INTRODUCTION

Many high performance computing (HPC) applications take a very long time to run even when using a large number of concurrent compute resources. Examples of applications that have historically required very long runtimes on HPC resources include molecular dynamics simulations, fluid-flow simulations, astrophysical compact object merger simulations, and mathematical optimization problems. Apart from these, an application that does not strong scale very well may require large runtimes because it can only effectively use a limited number of compute resources and would see no time-to-solution benefit when adding more. Applications with long execution times run a significant risk of encountering a hardware or software failure before completion. Long execution times also frequently violate supercomputer usage policies where a maximum wallclock time limit for a simulation is established to accommodate a large number of users better. In either case, the consequences of having a job killed can be very significant and costly in terms of time lost and computing resources wasted. Checkpointing is one way to help mitigate this risk.

At designated points during the execution of an application on a supercomputer, the data necessary to allow later resumption of the application at that point in the execution can be output and saved. This data is called a checkpoint, and the resumption of application execution is called a restart. It is no surprise that checkpoint files can be extremely large. Beyond mitigating the cost of an execution failure during a simulation that runs for a long time, checkpoint files provide snapshots of the application at different simulation epochs, help in debugging, aid in performance monitoring and analysis, and can help improve load-balancing decisions for better distributed-memory usage. This chapter explores two different approaches to checkpointing frequently encountered in HPC: system-level approaches and application-level approaches.

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00020-4

Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 576

20.2 SYSTEM-LEVEL CHECKPOINTING

System-level checkpointing performs the checkpoint and restart procedures via a full memory dump. This type of checkpointing does not require any changes to the application to enable its use, and writing of the checkpoint may be triggered either by the system or by the user. Examples of such user-transparent approaches for HPC support include Berkeley Lab Checkpoint/Restart [1], Checkpoint/Restore in Userspace [2], and Distributed MultiThreaded CheckPointing (DMTCP) [3]. These system-level approaches are generally fully integrated with the resource management system on a supercomputer, including Simple Linux Utility for Resource Management (SLURM) and Portable Batch System (PBS), and provide checkpoint/restart support for multithreaded applications and distributed-memory applications based on the message-passing interface (MPI). They are fully transparent to the user, requiring no changes to an application code, although they generally require a preload library step and inputs to specify the checkpoint interval, checkpoint directory, and restart directory.

The key advantage of system-level checkpoint/restart approaches over application-level approaches is that they require no changes to the application source code. Additionally, many system-level approaches incorporate access to kernel resource information, such as process IDs, which can simplify restarting the application. However, because the system-level checkpointing strategy includes a full memory dump, the checkpoint files may be significantly larger than just saving the smallest amount of relevant information, as is done with an application-level approach.

As an example of interactive system-level checkpointing, the OpenMP code in Code 20.1 is used in conjunction with the DMTCP tool in this section.

```
 1 #include <omp.h>
 2 #include <unistd.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5 #include <math.h>
 6
 7 int main (int argc, char *argv[])
 8 {
 9   const int size = 20;
10   int nthreads, threadid, i;
11   double array1[size], array2[size], array3[size];
12
13   // Initialize
14   for (i=0; i < size; i++) {
15     array1[i] = 1.0*i;
16     array2[i] = 2.0*i;
17   }
18
19   int chunk = 3;
20
21 #pragma omp parallel private(threadid)
22   {
23   threadid = omp_get_thread_num();
24   if (threadid == 0) {
```

```
25     nthreads = omp_get_num_threads();
26     printf("Number of threads = %d\n", nthreads);
27   }
28   printf(" My threadid %d\n",threadid);
29
30   #pragma omp for schedule(static,chunk)
31   for (i=0; i<size; i++) {
32     array3[i] = sin(array1[i] + array2[i]);
33     printf(" Thread id: %d working on index %d\n",threadid,i);
34     sleep(1);
35   }
36
37   } // join
38
39   printf(" TEST array3[199] = %g\n",array3[199]);
40
41   return 0;
42 }
```

Code 20.1. Example OpenMP code, checkpoint_openmp.c, for demonstrating system-level checkpointing. A "sleep" statement has been added to line 34 to add a pause to the execution after each thread performs the operation of line 32.

DMTCP provides several easy-to-use commands for transparent system-level checkpointing. The *dmtcp_coordinator* acts as a command-line interface to DMTCP for examining the checkpoint interval, accessing status messages, and forcing a manual checkpoint outside the specified checkpoint interval from the command line. The *dmtcp_coordinator* is launched in a separate terminal and awaits command-line input instructions and outputs status messages, as shown in Fig. 20.1.

```
andersmw@cutter:~/dmtcp-dmtcp-35386c2/bin$ ./dmtcp_coordinator
dmtcp_coordinator starting...
    Host: cutter (156.56.64.43)
    Port: 7779
    Checkpoint Interval: disabled (checkpoint manually instead)
    Exit on last client: 0
Type '?' for help.

?
COMMANDS:
  l : List connected nodes
  s : Print status message
  c : Checkpoint all nodes
  i : Print current checkpoint interval
      (To change checkpoint interval, use dmtcp_command)
  k : Kill all nodes
  q : Kill all nodes and quit
  ? : Show this message
```

FIGURE 20.1

The *dmtcp_coordinator* for status updates and interact with DMTCP via the specific commands listed here, including forcing a checkpoint outside the checkpoint interval by issuing the "c" command.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 578

To checkpoint the code illustrated in Code 20.1, it is compiled just as if it were not being checkpointed:

```
gcc -fopenmp -O3 -o checkpoint_openmp checkpoint_openmp.c -lm
```

The math library ("-lm") is added for the sin(x) function used on line 32 of Code 20.1 and the executable is named "checkpoint_openmp".

The number of OpenMP threads is also set in the normal way through the environment variable OMP_NUM_THREADS (illustrated here using bash shell syntax; for tcsh shell, use *setenv*):

```
export OMP_NUM_THREADS=16
```

The checkpoint interval can be changed using *dmtcp_command*, which sends the command to the *dmtcp_coordinator* already launched in Fig. 20.1:

```
dmtcp_command --interval <checkpoint interval in seconds>
```

Because Code 20.1 executes very quickly, a checkpoint request will be manually input into the *dmtcp_coordinator* command interface. The executable is launched with checkpoint capability using the *dmtcp_launch* tool:

```
dmtcp_launch ./checkpoint_openmp
```

The executable begins to run as normal, and if a checkpoint interval has been supplied, at every specified interval of wallclock time a checkpoint is written to the file system. Additionally, if the command "c" is supplied to the *dmtcp_coordinator* command interface, a checkpoint is written to the file system at that point. DMTCP checkpoint files have the naming convention of "ckpt_<executable name>_<client identity>.dmtcp" and are written in the directory where the executable was launched. A manually issued checkpoint request is illustrated in Fig. 20.2, which creates, in this example, a checkpoint file named *ckpt_checkpoint_openmp_16707112e4c8f-42000-8687a700c18a5.dmtcp*.

The checkpoint file is restarted using the *dmtcp_restart* command:

```
dmtcp_restart <checkpoint file>
```

A snippet of the standard output for Code 20.1 with and without checkpoint restart is shown in Fig. 20.3. The same OpenMP threads operate on the same array indices and all operations are identical in the restarted case and the nonrestarted case. No changes were made to the code to enable

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 579

```
c
[40367] NOTE at dmtcp_coordinator.cpp:1071 in startCheckpoint; REASON='starting checkpoint, suspending all nodes'
     s.numPeers = 1
[40367] NOTE at dmtcp_coordinator.cpp:1073 in startCheckpoint; REASON='Incremented computationGeneration'
     compId.computationGeneration() = 1
[40367] NOTE at dmtcp_coordinator.cpp:413 in updateMinimumState; REASON='locking all nodes'
[40367] NOTE at dmtcp_coordinator.cpp:419 in updateMinimumState; REASON='draining all nodes'
[40367] NOTE at dmtcp_coordinator.cpp:425 in updateMinimumState; REASON='checkpointing all nodes'
[40367] NOTE at dmtcp_coordinator.cpp:449 in updateMinimumState; REASON='building name service database'
[40367] NOTE at dmtcp_coordinator.cpp:465 in updateMinimumState; REASON='entertaining queries now'
[40367] NOTE at dmtcp_coordinator.cpp:470 in updateMinimumState; REASON='refilling all nodes'
[40367] NOTE at dmtcp_coordinator.cpp:510 in updateMinimumState; REASON='restarting all nodes'
```

FIGURE 20.2

A manually issued checkpoint request followed by the associated status messages from DMTCP for checkpointing Code 20.1.

```
andersmw@cutter:~/textbook$ dmtcp_restart ckpt_checkpoint_openmp_16707112e4c8f-42000-8687a700c18a5.dmtcp
 Thread id: 15 working on index 141
 Thread id: 14 working on index 138
 Thread id: 7 working on index 117
 Thread id: 12 working on index 132
 Thread id: 1 working on index 99
 Thread id: 2 working on index 102
 Thread id: 8 working on index 120
 Thread id: 5 working on index 111
 Thread id: 11 working on index 129
 Thread id: 13 working on index 135
 Thread id: 3 working on index 105
 Thread id: 10 working on index 126
 Thread id: 9 working on index 123
 Thread id: 6 working on index 114
 Thread id: 4 working on index 108
 Thread id: 0 working on index 96
```

```
Thread id: 15 working on index 141
Thread id: 14 working on index 138
Thread id: 8 working on index 120
Thread id: 1 working on index 99
Thread id: 2 working on index 102
Thread id: 4 working on index 108
Thread id: 12 working on index 132
Thread id: 7 working on index 117
Thread id: 0 working on index 96
Thread id: 13 working on index 135
Thread id: 11 working on index 129
Thread id: 5 working on index 111
Thread id: 3 working on index 105
Thread id: 9 working on index 123
Thread id: 6 working on index 114
Thread id: 10 working on index 126
```

FIGURE 20.3

The standard output from Code 20.1 after checkpoint restart (left) and without restart (right). The same OpenMP threads operate on the same indices and all operations are identical in the restarted case and the nonrestarted case. As is standard in system-level checkpointing, no changes to Code 20.1 were made to enable checkpoint capability.

checkpoint/restart capability, and the checkpoint files written could also be used for debugging, execution snapshots, or as part of a strategy for fault tolerance.

Interactive system-level checkpoint/restart using DMTCP for an MPI application is similar to that for an OpenMP application, but with small differences. An example MPI "pingpong" code, referred to as pingpong.c, using *MPI_Send* and *MPI_Recv* is shown in Code 20.2, which passes back and forth an integer and increments that integer for each iteration.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include "mpi.h"
5
6 int main(int argc,char **argv)
7 {
8   int rank,size;
9   MPI_Init(&argc,&argv);
10  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11  MPI_Comm_size(MPI_COMM_WORLD,&size);
12
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 580

```
13   if ( size != 2 ) {
14     printf(" Only runs on 2 processes \n");
15     MPI_Finalize();    // this example only works on two processes
16     exit(0);
17   }
18
19   int count;
20   if ( rank == 0 ) {
21     // initialize count on process 0
22     count = 0;
23   }
24   for (int i=0;i<10;i++) {
25     if ( rank == 0 ) {
26       MPI_Send(&count,1,MPI_INT,1,0,MPI_COMM_WORLD); // send "count" to rank 1
27       MPI_Recv(&count,1,MPI_INT,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE); // receive it
         back
28       sleep(1);
29       count++;
30       printf(" Count %d\n",count);
31     } else {
32       MPI_Recv(&count,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
33       MPI_Send(&count,1,MPI_INT,0,0,MPI_COMM_WORLD);
34     }
35   }
36
37   if ( rank == 0 ) printf("\t\t\t Round trip count = %d\n",count);
38
39   MPI_Finalize();
40 }
```

Code 20.2. Example MPI "pingpong" code for demonstrating system-level checkpoint/restart using DMTCP. A "sleep" command has been added to line 28 to slow down the execution for checkpoint demonstration purposes. This code is designed to work on only two processes and will print out the "count" integer at each message epoch.

Just as in the OpenMP checkpoint/restart example, the code is not modified and is compiled as usual without including any extra libraries specific to checkpoint/restart:

```
mpicc -O3 -o pingpong pingpong.c
```

In this example MPICH-2 is the MPI implementation used; DMTCP supports several different implementations of MPI. After the *dmtcp_coordinator* is started in a separate window to issue manual checkpoint commands and monitor status messages, the pingpong executable is then launched on two processes using a combination of *dmtcp_launch* and *mpirun* as follows:

```
dmctp_launch --rm mpirun -np 2 ./pingpong
```

```
c
[22984] NOTE at dmtcp_coordinator.cpp:1071 in startCheckpoint; REASON='starting checkpoint, suspending all nodes'
     s.numPeers = 4
[22984] NOTE at dmtcp_coordinator.cpp:1073 in startCheckpoint; REASON='Incremented computationGeneration'
     compId.computationGeneration() = 1
[22984] NOTE at dmtcp_coordinator.cpp:413 in updateMinimumState; REASON='locking all nodes'
[22984] NOTE at dmtcp_coordinator.cpp:419 in updateMinimumState; REASON='draining all nodes'
[22984] NOTE at dmtcp_coordinator.cpp:425 in updateMinimumState; REASON='checkpointing all nodes'
[22984] NOTE at dmtcp_coordinator.cpp:449 in updateMinimumState; REASON='building name service database'
[22984] NOTE at dmtcp_coordinator.cpp:465 in updateMinimumState; REASON='entertaining queries now'
[22984] NOTE at dmtcp_coordinator.cpp:470 in updateMinimumState; REASON='refilling all nodes'
[22984] NOTE at dmtcp_coordinator.cpp:510 in updateMinimumState; REASON='restarting all nodes'
```

FIGURE 20.4

Status messages generated after issuing the checkpoint command ("c") to the *dmtcp_coordinator*. Each process generates a checkpoint file, which is stored in the directory where the executable was launched.

After five message epochs the command for generating the checkpoint ("c") is issued to the *dmtcp_coordinator*, as illustrated in Fig. 20.4.

Four checkpoint files result from the checkpoint command, one from each process and two associated with the MPI launcher. A restart script specific to the checkpoint files generated is also created to simplify the restart process. This script is created in the directory where the *dmtcp_coordinator* was launched and is called *dmtcp_restart_script_<client identity>.sh*. The script requires no arguments and already knows where to find the checkpoint files in the file system. Launching this shell script will restart the job, as illustrated in Fig. 20.5.

Both the OpenMP and MPI examples explored here using the DMTCP system-level checkpointing tool were performed interactively for ease of demonstration. However, on most supercomputing systems a user does not attempt to perform a checkpoint/restart interactively but launches applications through a resource management system like PBS or SLURM. DMTCP, like the other system-level checkpointing tools mentioned here, is integrated with PBS and SLURM and provides example scripts for launching and restarting applications through a resource management system. In the case of DMTCP, using a resource management system to checkpoint an MPI or OpenMP application requires the *dmtcp_coordinator* to be launched as a daemon in the PBS or SLURM script while the other commands (*dmtcp_launch, dmtcp_restart_script)* remain the same, as was demonstrated in interactive mode. On HPC resources with an Infiniband network, the *dmtcp_launch* command also requires the flag *–infiniband* for checkpoint/restart support of MPI-based applications using Infiniband.

```
[andersmw@cutter textbook]$ ./dmtcp_restart_script_129b065bca8bba11-56000-87bae5e9cb04.sh
 Count 6
 Count 7
 Count 8
 Count 9
 Count 10
                    Round trip count = 10
```

```
Count 1
Count 2
Count 3
Count 4
Count 5
Count 6
Count 7
Count 8
Count 9
Count 10
                    Round trip count = 10
```

FIGURE 20.5

The standard output from the MPI "pingpong" from Code 20.2 after checkpoint restart (left) and without using any checkpoint/restart (right). The checkpoint restart case (left) began from checkpoint data generated after the fifth epoch, and consequently the first output seen after restart is the sixth epoch.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 582

20.3 APPLICATION-LEVEL CHECKPOINTING

In application-level checkpointing the application developer has the responsibility to perform all checkpoint/restart operations. As opposed to system-level checkpointing, application-level checkpointing requires changes to the application code. While inconvenient, application-level checkpoint/restart tends to produce checkpoint files that are smaller than system-level checkpoint/restart, where a full memory dump is performed. Checkpoint files originating from application-level checkpointing are generally smaller than those originating from system-level checkpointing approaches simply because the application developer will only output the most pertinent information necessary for application restart. The system, in contrast, has to dump the entire application memory because it cannot single out what data is relevant for restart.

For distributed-memory applications based on MPI, application-level checkpoint/restart approaches often share some basic characteristics:

- Only one checkpoint file is written per MPI process.
- Only one MPI rank accesses a single checkpoint file.
- Checkpoint files do not contain data from multiple checkpoint epochs.
- Checkpoint files are generally written to the parallel file system by the compute nodes.
- Checkpoint/restart overheads can be large.

Application-level checkpoint/restart implementations generally pick designated points in the computational phase in the simulation algorithm for checkpointing, to ensure computational phase consistency in the checkpoint epoch. For example, in a timestepping algorithm a natural place to incorporate checkpoint/restart would be at the end of one timestep, thereby ensuring that all checkpoint files are at the same computational phase even if they each reached this phase at different wallclock times. This is in contrast to system-level checkpointing, where, regardless at what phase of computation the process may be, a checkpoint is dumped as designated by a wallclock time interval or an event such as a manual request for checkpoint given in the command line. Consequently, application-level checkpoint/restart implementations may not specify a checkpoint interval in terms of wallclock time, as in system-level approaches, but rather require the interval of computational phases for checkpoint/restart.

Some of the I/O libraries explored in Chapter 10 are especially well suited for use in application-level checkpointing. For instance, the HDF5 library is widely used in this checkpointing because it is well suited for parallel I/O and creates data structured for different execution configurations as well as providing portability. As with any parallel I/O operation, however, the developer will still have to ensure that all application data is actually written to the checkpoint files and not just the pointer addresses to data. Because C codes frequently access data indirectly between different functions, it is a common novice C programmer mistake to output a pointer address rather than the data itself.

Application-level checkpoint/restart is very popular in large-scale MPI applications and toolkits because it can be tailored for the application to be as efficient and minimal as possible. However, the checkpoint/restart overhead is still very high and there are checkpoint/restart libraries written to assist in reducing this overhead for application-level checkpoint approaches. One such library is Scalable Checkpoint/Restart (SCR) for MPI [4].

The SCR library assists application-level checkpoint strategies by reducing the load on a parallel file system and partially utilizing nonparallel fast storage local to a compute node for checkpoint file

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 583

storage, with some redundancy in the event of a failure on the local storage. SCR provides several different checkpoint file redundancy schemes with varying levels of resilience and performance. It requires the parallel remote shell command [5] and the Perl module for date/time interpretation [6], and works natively with the SLURM resource manager.

The SCR library is built around an application-level checkpoint strategy like that illustrated in Code 20.3 where only one checkpoint file is written by an MPI process. When using the SCR library, the library needs to know when to start a checkpoint and when to finish a checkpoint through API calls that are collective across all MPI processes. The SCR library can also determine if a checkpoint file is needed rather than having some user-defined checkpoint frequency, as was seen in Code 20.3. This is done by configuring SCR with system information to estimate checkpoint costs and frequency of failure, and then using the application programming interface (API) call *SCR_Need_checkpoint* to let SCR decide the frequency of checkpointing.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "mpi.h"
5
6
7 int write_checkpoint()
8 {
9   // get our rank
10   int rank;
11   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12
13   char file[128];
14   sprintf(file,"checkpoint/%d_checkpoint.dat",rank);
15
16   FILE *fp = fopen(file,"w");
17
18   // write sample checkpoint to file
19   fprintf(fp," Hello Checkpoint World\n");
20   fclose(fp);
21
22   return 0;
23 }
24
25 int main(int argc,char **argv)
26 {
27   MPI_Init(&argc,&argv);
28
29   int max_steps = 100;
30   int step;
31   int checkpoint_every = 10;
32
33   for (step=0;step<max_steps;step++) {
34     /* perform simulation work */
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 584

```
35
36     if ( step%checkpoint_every == 0 ) {
37       write_checkpoint();
38     }
39   }
40
41   MPI_Finalize();
42   return 0;
43 }
```

Code 20.3. Simple example of a common application-level checkpoint strategy. Each process writes its own checkpoint data to a single checkpoint file. The frequency of writing the checkpoint is determined by the user, set here to be 10 (line 31).

The modifications to Code 20.3 needed to incorporate the SCR library are limited to adding the calls *SCR_Init*, *SCR_Finalize*, *SCR_Start_checkpoint*, *SCR_Complete_checkpoint*, and *SCR_Route_file*. Optionally, the checkpoint frequency can be determined by SCR using the call *SCR_Need_checkpoint*, as already noted. *SCR_Init* and *SCR_Finalize* initialize and shut down the SCR library, analogous to *MPI_Init* and *MPI_Finalize*. *SCR_Start_checkpoint* and *SCR_Complete_checkpoint* indicate, respectively, that a checkpoint is about to begin to write and a checkpoint has successfully been written. *SCR_Route_file* is used for getting the full path and file name for SCR access. Each SCR API call is collective across all MPI processes. The SCR version of Code 20.3 is provided in Code 20.4.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "scr.h"
4 #include "mpi.h"
5
6 int write_checkpoint()
7 {
8   SCR_Start_checkpoint();
9
10  // get our rank
11  int rank;
12  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13
14  char file[128];
15  sprintf(file,"checkpoint/%d_checkpoint.dat",rank);
16
17  FILE *fp = fopen(file,"w");
18
19  char scrfile[SCR_MAX_FILENAME];
20  SCR_Route_file(file,scrfile);
21
22  // write sample checkpoint to file
23  fprintf(fp," Hello Checkpoint World\n");
24  fclose(fp);
25
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 585

```
26    int valid = 1;
27
28    SCR_Complete_checkpoint(valid);
29
30    return 0;
31  }
32
33  int main(int argc,char **argv)
34  {
35    MPI_Init(&argc,&argv);
36
37   if ( SCR_Init() != SCR_SUCCESS ) {
38     printf(" SCR didn't initialize\n");
39     return -1;
40   }
41
42   int max_steps = 100;
43   int step;
44   for (step=0;step<max_steps;step++) {
45     /* perform simulation work */
46
47     int checkpoint_flag;
48     SCR_Need_checkpoint(&checkpoint_flag);
49     if ( checkpoint_flag ) {
50        write_checkpoint();
51     }
52   }
53   SCR_Finalize();
54   MPI_Finalize();
55   return 0;
56 }
```

Code 20.4. SCR version of Code 20.3 application-level checkpointing. Calls to the SCR API include SCR_Init (line 37), SCR_Finalize (line 53), SCR_Start_checkpoint (line 8), SCR_Complete_checkpoint (line 28), SCR_Need_checkpoint (line 48), and SCR_Route_file (line 20). The SCR_Need_checkpoint call is optional and allows SCR to control the checkpoint frequency. Relatively few changes are needed to an existing application-level checkpoint strategy to take advantage of the benefits of SCR.

To use SCR and execute Code 20.4, SCR must be integrated with the supercomputer's resource management system. In the case of SLURM, an SCR-enabled code would launch using *scr_srun* instead of *srun*.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 586

20.4 SUMMARY AND OUTCOMES OF CHAPTER 20

- Applications with long execution times run a significant risk of encountering a hardware or software failure before completion.
- Long execution times also frequently violate supercomputer usage policies where a maximum wallclock limit for a simulation is established.
- The consequences of a hardware or software failure can be very significant and costly in terms of time lost and computing resources wasted for long-running jobs.
- At designated points during the execution of an application on a supercomputer, the data necessary to allow later resumption of the application at that point in the execution can be output and saved. This data is called a checkpoint.
- Checkpoint files help mitigate the risk of a hardware or software failure in a long-running job.
- Checkpoint files also provide snapshots of the application at different simulation epochs, help in debugging, aid in performance monitoring and analysis, and can help improve load-balancing decisions for better distributed-memory usage.
- In HPC applications, two common strategies for checkpoint/restart are employed: system-level checkpoint and application-level checkpointing.
- System-level checkpointing requires no modifications to the user code but may require loading a specific system-level library.
- System-level checkpointing strategies center on full memory dumps and may result in very large checkpoint files.
- Application-level checkpointing requires modifications to the user code. Libraries exist to assist this process.
- Application-level checkpoint files tend to be more efficient, since they only output the most relevant data needed for restart.

20.5 EXERCISES

1. List the trade-offs between system-level checkpointing and application-level checkpointing. Survey the some of the many scientific computing toolkits available for download that have checkpoint/restart capability. What form of checkpointing is the most popular in these toolkits?
2. How might checkpoint files be used for debugging? Illustrate this by introducing a race condition into Code 20.1, such as a reduction operation without the appropriate reduction clause, and expose the bug by using a checkpoint file. Use system-level checkpointing.
3. For an application that runs on 100,000 cores for 9 days of wallclock time, estimate the likelihood that the application will encounter a hardware failure during the simulation. Use the reported annualized failure rate for a hypothetical collection of hard drives, processors, and power supplies.
4. What could happen if a system failure occurs while a checkpoint is being written? What are the ways to mitigate this type of failure?
5. What is the trade-offs between checkpointing frequently and infrequently? Suppose the example in Code 20.2 is checkpoint every 1 s versus every 30 s. What are the performance consequences and benefits of these?

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 587

REFERENCES

[1] Berkely Laboratory, Berkeley Lab Checkpoint/Restart (BLCR) for LINUX. [Online] http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/.
[2] CRIU, Checkpoint/Restore In Userspace. [Online] https://criu.org/.
[3] DMTCP: Distributed MultiThreaded CheckPointing. [Online] http://dmtcp.sourceforge.net/.
[4] Lawrence Livermore National Laboratory, Scalable Checkpoint/Restart Library. [Online] http://computation.llnl.gov/projects/scalable-checkpoint-restart-for-mpi/software.
[5] Parallel Remote Shell Command (PDSH). [Online] http://sourceforge.net/projects/pdsh.
[6] Perl Date Maniputation. [Online] http://search.cpan.org/~sbeck/Date-Manip-6.56.

CHAPTER

NEXT STEPS AND BEYOND

21

CHAPTER OUTLINE

21.1 Introduction **606**
21.2 Expanded Parallel Programming Models **606**
21.2.1 Advance in Message-Passing Interface 606
21.2.2 Advances in OpenMP 607
21.2.3 MPI+X 607
21.3 Extended High Performance Computing Architecture **608**
21.3.1 The World's Fastest Machine 608
21.3.2 Lightweight Architectures 608
21.3.3 Field Programmable Gate Arrays 609
21.4 Exascale Computing **610**
21.4.1 Challenges to Exascale Computing 611
21.4.2 Doing the Math, How Big Is Exascale? 611
21.4.3 The Accelerated Approach 612
21.4.4 Lightweight Cores 612
21.5 Asynchronous Multitasking **613**
21.5.1 Multithreaded 613
21.5.2 Message-Driven Computation 613
21.5.3 Global Address Space 614
21.5.4 Actor Synchronization 615
21.5.5 Runtime System Software 615
21.6 The Neodigital Age **616**
21.6.1 Dataflow 617
21.6.2 Cellular Automata 618
21.6.3 Neuromorphic 619
21.6.4 Quantum Computing 619
21.7 Exercises **620**
References **621**

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00021-6
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 589

21.1 INTRODUCTION

This textbook gives a comprehensive top-level coverage of high performance computing (HPC), both to present the complex interdisciplinary components of the field and to provide the basic skill-sets that an entry-level student practitioner requires and can use to employ such systems for end-user applications. It presents major classes of architecture, programming models, basic algorithmic techniques, and widely used tools and environments. More deeply, it shares the fundamental concepts that govern the challenges of efficiency, scalability, parallel semantics, and metrics. But while an excellent first coverage of the broad field, it is far from complete; it is more of a starting point than exhaustive. In this chapter the authors complete this treatise by noting what has not been covered that may be of interest and serve as a roadmap for further study in specific areas, incrementally building on what has already been presented. This is done in two domains. Sections 21.2 and 21.3 describe in brief the more sophisticated techniques currently employed in both programming models and hardware architectures. Section 21.4 discusses current directions toward exascale computing, which is occupying much energy in near-term research across the northern hemisphere, with likely impact in the early 2020s. Section 21.5 considers a shift in computing methods being explored, sometimes referred to as "asynchronous multitasking", that will enable dynamic adaptive techniques for improved efficiency and scalability. Section 21.6 on the "neodigital age" may be viewed for your curiosity. It projects ideas about what may happen with the end of Moore's law and nanoscale semiconductor technology, and where revolutionary approaches to computer architecture going beyond the von Neumann architecture and its decades of derivatives may take us. This includes "quantum computing", which is in a most inchoate phase, but if it proves possible will be able to perform some computations that could not be done even in the lifetime of the universe if performed on even the biggest conventional computer.

21.2 EXPANDED PARALLEL PROGRAMMING MODELS

21.2.1 ADVANCE IN MESSAGE-PASSING INTERFACE

The message-passing interface (MPI) is one of the most widely used means of describing programs to run on scalable distributed-memory systems comprising multiple computing nodes integrated via one or more interconnection networks. This textbook describes the programming principles of MPI consistent with the MPI-1 standard, including the basics of establishing virtual topologies among processes, send/receive message communication constructs, scalar data types and some complex data structures, collective operations for synchronization, data distribution, and collective reduction operations. But in total only a couple of dozen MPI functions are described. These are sufficient to represent a wide range of useful parallel algorithms and run these applications on a wide range of large-scale current-generation systems, yet only scratch the surface of the rich set of commands actually available even by MPI-1 for optimization and to facilitate sophisticated communication and shared computation patterns. Since the final specification of MPI-1 in 2008, including more than 120 functions, MPI has evolved as a model and a parallel programming interface, expanding to the advanced versions of MPI-2 and MPI-3. MPI-2 made important extensions to the original standard,

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 590

including a strong set of input/output (I/O) calls to manage access to mass storage, dynamic process management, and single-sided functions on remote memory. MPI-3 added significant extensions to the previous versions, expanding the full set of collective operations specifically in the domain of non-blocking as well as other improvements.

21.2.2 ADVANCES IN OPENMP

OpenMP has been demonstrated as a popular programming interface for shared-memory computing systems. First introduced in 1997 with Fortran bindings and in 2002 with C/C++ bindings for OpenMP-2, OpenMP language extensions have provided environment variables, directives, and library functions for transforming sequential application codes into programs that include multiple-thread computing for a degree of parallelism and reduction in time to solution. This textbook presents the foundation concepts of OpenMP, and many of the key constructs and their syntax. The full OpenMP specification is far larger, with many valuable optimizations that are not covered. But even the chosen subset shows that substantive and diverse applications can be constructed and run on a wide array of systems. Further advances beyond the basic functions were devised in later versions of OpenMP. For example, the powerful capability of multitasking and the task construct were a centerpiece of OpenMP-3 in 2008. In 2013 OpenMP-4 was released with a number of significant improvements, such as support for heterogeneous systems incorporating accelerators, the inclusion of thread affinity to assist in managing some aspects of locality, and methods for exploiting single-instruction multiple data (SIMD) parallelism. OpenMP has been very successful, but has some shortcomings that bound its effectiveness in terms of efficiency and scalability. Because it assumes a shared-memory ecosystem, it is limited in scaling on single symmetric multiprocessor (SMP) systems. This is in part mitigated by increasing the number of cores integrated on chip and per socket. Its efficiency is constrained by its heavy use of fork—join semantics in which global barriers play a significant part. This is sensitive to Amdahl's law with the purely sequential parts of any OpenMP code. The tasking mechanisms can help offset this property.

21.2.3 MPI+X

In brief, MPI provides a form of scalability with coarse-grain parallelism across distributed-memory systems while OpenMP provides a form of medium-grain parallelism within the boundaries of shared-memory nodes. Neither is sufficient for the future challenges of exascale computing and beyond. But the combination of the two, each complementing the other, is viewed as a significant opportunity for the next stages in the field. MPI processes across distributed-memory system architectures will continue to provide the scalability required, while OpenMP delineates medium-grain threads that can be performed by the many cores within a single node. This permits a coarse-grained MPI process to span the entire node (as it did in the early days of massively parallel processors—MPPs), but allows the efficiencies of shared memory hardware to be exploited and greater parallelism to be exposed with the assistance of OpenMP constructs. The general concept has been referred to as "MPI+X", where "X" refers to another or additional programming interface working in cooperation with MPI. X could also mean OpenCL, or perhaps even configuring field programmable gate arrays (FPGAs) within a node. As Bill Gropp said, "The important part of MPI+X is the '+'." [1]

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 591

21.3 EXTENDED HIGH PERFORMANCE COMPUTING ARCHITECTURE

The end of Moore's law marks a milestone in HPC. Over multiple decades technology could be anticipated to deliver exponential growth in device density and clock rate. But this is rapidly changing, with parallel architecture being the only remaining strategy to continue performance growth, at least until some significant advance in enabling technology emerges to replace CMOS (such as superconducting Josephson junction logic) or an entirely new paradigm (such as quantum computing) is developed and made practical. This section touches on some advances in HPC system architecture that are being pursued.

21.3.1 THE WORLD'S FASTEST MACHINE

The world's fastest machine as of June 2017 is the third Chinese machine to make the number 1 slot of the Top 500 list in recent years: the Sunway TaihuLight (神威·太湖之光) based on Sunway microprocessors. It is an example of pursuing alternative approaches outside conventional architectures. This system, which has a peak performance greater than 100 Petaflops and a Linpack rating of almost that much, is based on a new architecture fully developed and manufactured in China. The architecture features an extremely lightweight core that dispenses with many of the conventional internal subcomponents such as data caches. While some efficiency, as measured by arithmetic logic unit utilization, is degraded, a much larger number of cores can be integrated on to a die, each using much less energy. This machine, although more than two times faster than its predecessor (also Chinese), uses less than half as much electrical power. This is a remarkable achievement. The Sunway architecture is also controversial, as its memory capacity is relatively small given its peak floating-point performance. But in a very short time it has demonstrated significant achievement in real-world applications. In all it has more than 10 million cores—an unprecedented record.

21.3.2 LIGHTWEIGHT ARCHITECTURES

The vast majority of the fastest supercomputers (see the Top 500 list) employ either Intel ×86 architecture microprocessors (including AMD variants) or IBM Power-based microprocessors with or without accelerators, including Nvidia and Intel Phi. However, there is a trend toward lightweight architectures to increase peak performance per socket and reduce power consumption and space costs.

As noted, a dramatic example is the TaihuLight, the world's fastest supercomputer measured by the high performance Linpack (HPL) benchmark in 2017. It comprises 10 million cores, each of which is very lightweight with only some scratchpad high-speed memory (with an instruction cache). Sixty-four of these are organized in a "group", and there are four groups in a processor socket for 256 cores plus four management processing elements to manage the computation. At a peak performance of 125 petaflops it is the first system to enter the 100 Pflops performance regime.

A second important trend is the evolution of the Intel Xeon Phi processor, derived from the failed Larrabee Project [2]. Intel chose to address the challenge of ultrahigh performance through pervasive integration of a semiconductor die with many lightweight cores rather than the fewer heavyweight

Xeon cores that have dominated MPPs and commodity clusters for 2 decades. These were originally treated like graphics processing units (GPUs), as attached processors with printed circuit boards (PCBs) incorporating Phi chips integrated via industry interface standards, principally PCI express (PCIe). While a convenient way to introduce a new technology rapidly, this approach suffered from the treatment of these devices as attached array processors controlled by master processors and separated by relatively slow Peripheral Component Interconnect (PCI) buses in terms of latency, bandwidth, and control overheads. The latest generation of Xeon Phi corrects these shortcomings by allowing the Phi processor sockets to operate in "self-hosting" mode; that is, to be their own masters, eliminating the PCI bottlenecks and equally important the control overheads. This is a big deal, and will be first demonstrated in the 2018 as currently planned.

An alternative path that is trending is the evolution of the long-serving ARM processor architecture. The heritage of ARM is in the vast operational space of mobile, embedded, and control processing. ARM can be custom configured by the end implementer to provide a wide variation in the ecosphere of the support logic and interfaces on die. Although primarily a 32-bit architecture, ARM has now been extended to several variants of 64-bit architecture, making it truly suitable in the context of supercomputing for conventional numeric applications such as simulation and data analysis. At least one major-scale experimental supercomputer, Mont Blanc, is under development in Europe using ARM as the principal processor core. Unconfirmed reports suggest that the National University of Defense Technology in China is pursuing a similar approach. While ARM is not currently regarded as part of the HPC field, this may change radically toward the end of this decade.

21.3.3 FIELD PROGRAMMABLE GATE ARRAYS

An FPGA is, as the name implies, a component comprising a large number of logic gates and other functional parts connected by a network, the connectivity of which can be determined by "programming" the device. That is, there is a protocol by which the end user can determine the logic circuitry of the component. While less dense and somewhat slower than application-specific integrated circuits (ASICs), FPGAs enable custom designs to be produced to optimize them for special-purpose functionality. This permits the rapid development of prototype designs and gives a means to distribute a small number of parts to end users. One area of use that may prove promising is application-specific FPGA logic circuits optimized for specific algorithms. Such structures as systolic arrays can be implemented readily with FPGAs to accelerate important applications by one to two orders of magnitude with respect to conventional microprocessors. Other uses may include logic designs to support future system software to reduce overheads.

The major challenge is to provide efficient functionality that best suits application algorithms and the means of rapidly programming FPGAs. Much work has been done in both domains, but use still demands expertise. Another problem is the integration of FPGAs with otherwise conventional systems. This is in part addressed through industry-standard interfaces to which custom boards are designed with FPGA components. But this still has its limitations. Now hybrid subsystems with both processors and FPGAs integrated together are being made available, again improving their mutual connectivity.

21.4 EXASCALE COMPUTING

IBM Blue Gene/Q. *Courtesy Argonne National Laboratory via Wikimedia Commons*

Alan Gara is the chief architect of the highly successful series of IBM Blue Gene supercomputers, named after its intended applications: study of gene development and protein folding. Blue Gene architecture was a significant departure from the then leading Earth Simulator computer emphasizing vector processing. Blue Gene instead incorporated a large number of simple cores derived from embedded processors, resulting in improved energy efficiency. Its first model, Blue Gene/L, employed dual processor nodes with compute logic and NIC integrated on a single ASIC. The central processing units (CPUs) were based on PowerPC 440 with added double-precision floating-point pipelines delivering a peak of 5.6 Gflops per node. Thanks to high-density packaging, a single rack contained 1024 such nodes. Blue Gene/L (shown in the right picture) included 32,768 cores and debuted as number 1 on the Top 500 list in November 2004, achieving over 70 Tflops or twice the performance of the previous incumbent. In an updated configuration Blue Gene maintained this position for the next 3.5 years, achieving a peak throughput of nearly 600 Tflops while consuming only 2.3 MW of electricity. The architecture is also notable for incorporation of three network types: three-dimensional torus for point-to-point communication, a dedicated collective communication interconnect, and a global interrupt network. The later versions of Blue Gene architecture included P revision with quad-core nodes and a performance-to-power metric of 0.35 Gflops/W and Q, utilizing 18-core four-way simultaneously multithreaded processors and scaling to 20 Pflops.

For his work on three generations of Blue Gene architecture, Alan Gara was recognized with the Seymour Cray Award in 2010. He also codeveloped high performance implementations of quantum chromodynamics applications; one on the QCDSP custom architecture and the other on Blue Gene/L, each of which won the Gordon Bell Prize.

Historically within the field of supercomputing there has been a natural tendency of the community as a whole to speculate, consider, and ultimately to achieve the next three orders of magnitude performance-gain milestone over the previous major one. The first megaflops computer was the CDC-6600 in 1968, followed by the first gigaflops computer, the Cray-YMP in 1988, the

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 594

first teraflops machine, the Intel Red Storm in 1997, deployed at Sandia National Laboratories, and finally the first petaflops machine in 2008, the IBM Roadrunner deployed at Los Alamos National Laboratory. Roughly speaking these accomplishments occur approximately every 11 years. But this would suggest that the next milestone, exaflops, should be anticipated for 2019. While conceivable, this is not likely for a number of reasons. In recent years the fastest supercomputers measured by the HPL benchmark have been developed and deployed in China, with the most recent approaching 100 petaflops Rmax.

21.4.1 CHALLENGES TO EXASCALE COMPUTING

While most HPC systems within the mainstream are capable of one petaflops or less, the goal of exascale computing suggests 1000 times this norm, or more than 10 times the fastest computer in the world today. The challenges to achieving this are many and are application dependent. While technology is approaching the asymptote with respect to Moore's law and nanoscale semiconductors, exascale computing is still within the scope of Moore's law, even if approaching the end. The principal challenges as viewed by the industry and its users include the following:

- Energy and power—this is a limiting factor that goes beyond just the cost of the energy, which is about $1M/MWyear. It is also constrained by the maximum power that can feed a semiconductor die before it reaches a threshold of failure. A target goal is 20 MW or 50 Gflops/W.
- Hardware parallelism—expected to be in the order of a billion, which may be manifest by hundreds of millions of cores, each operating with 10-way parallelism like SIMD or vector.
- Software parallelism—application programs and algorithms using and exploiting more than a billionfold parallelism to take advantage of the hardware, including communication and secondary storage access.
- Overhead—the work required to manage the system and control each task. While a source of loss of efficiency, it also bounds the granularity of the tasks and therefore the available useful concurrency.
- Latency—with larger systems including more racks the physical distances for global access of data or services increase, requiring even more parallelism to hide the latency.
- Reliability—with the increase in the number of devices both on and off the die the chance of a single point failure increases and the potential reduction of the mean time to failure could make exascale computing impractical. Methods are required to provide resilience of both hardware and software with sufficient confidence that large and time-consuming computations can be performed.

21.4.2 DOING THE MATH, HOW BIG IS EXASCALE?

By any measure, exascale computing is enormous and its achievement, anticipated sometime in the next decade, will be a *tour de force*. In this textbook we have identified and considered a number of different dimensions by which a class of system can be measured. As a benchmark, the TaihuLight

Chinese system achieves about 100 petaflops with 10 million cores (a little less for Rmax, a little more for peak). This suggests that an exascale machine will require at least a 100 million cores for an exaflops. Of course these are lightweight cores with limited capability. Fewer cores would be required but at much more die space if heavyweight cores like the IBM Power 9 are employed, such as in the future (2018) Summit machine expected to operate at about 200 petaflops. But for real-world codes rather than for a friendly Linpack benchmark, much more will be required. With typical efficiencies around 10% (often less but sometimes more), this would project that there is significant room for dramatic improvements in delivered performance and reduced energy consumption.

21.4.3 THE ACCELERATED APPROACH

GPU accelerators are extremely effective for performing specific classes of streaming processing, and doing so with high-throughput performance. They incorporate many specialized processor cores interconnected to form useful dataflow paths to minimize the need for write-back of intermediate data and avoid control overheads. System nodes are heterogeneous, combining multicore CPU chips and GPU modules to permit computing workflows as appropriate running on the GPUs and the remaining computations being performed on the CPUs. An important path to exascale is a heterogeneous system architecture combining CPUs with GPUs which will provide high-density peak floating-point operations. A major challenge, programming such heterogeneous systems, is receiving significant attention, as is discussed in this book. The Summit supercomputer is planned to employ this kind of heterogeneous system architecture using GPUs; it will be deployed in 2018 at Oak Ridge National Laboratory with a delivered performance of approximately 200 petaflops.

21.4.4 LIGHTWEIGHT CORES

The alternative approach to achieving exascale performance is the use of a very large number of very lightweight cores. A typical core such as an Intel Xeon processor or IBM Power 8 architecture engages many mechanisms to keep the execution pipeline full and the time to execution of a thread to its minimum possible with the enabling technology. One school of thought is that to build the fastest system, one needs the fastest node; and to build the fastest node, the fastest core possible is required. But for a given socket package size and energy budget, a different strategy is to provide the highest number of cores possible, which means implementing the smallest-size core possible with full functionality. This was first tried with some success by IBM's Blue Gene systems using the lightweight PowerPC processor. Today Intel provides the Xeon Phi lightweight processor core with 76 cores per socket for Knights Landing. The next generation of Phi will be Knights Hill, which will provide the basis for the future systems to be deployed possibly in 2018. TaihuLight has 256 very lightweight cores per socket; a total of 10 million cores providing roughly 100 petaflops and using low energy. The ARM processor is emerging as yet another choice for large systems based on lightweight cores. Japan, China, and the European Union are all planning ARM-based systems of between 100 Petaflops and 1 Exaflops.

21.5 ASYNCHRONOUS MULTITASKING

Asynchrony is the property of uncertainty of timing and ordering of known events and operations. Greater scaling, such as remote data access or services performed, aggravates the variability of timing. Thus handling the effects of asynchrony becomes increasingly important as system scale and heterogeneity increase. A class of computing methodology referred to as "asynchronous multitasking" (AMT) is a topic of extensive research that shifts computing from static scheduling and conventional resource management methods to dynamic adaptive control of program execution and the application of available memory and processing resources. The following subsections describe aspects being considered as means for addressing the challenges imposed by asynchrony and exploiting the opportunities that asynchrony offers.

21.5.1 MULTITHREADED

Threads are generally considered as sequences of instructions sharing intermediate result data that can be scheduled on individual cores. Multithreaded computing is when there is more than one thread operating concurrently and possibly in parallel, in which case time to solution improves, perhaps proportionally with the number of threads. This is not a new concept, of course, but how it is implemented in terms of control, synchronization, scheduling, and resource allocation has evolved and differs substantially among different methodologies. Typically there is a one-to-one mapping of application threads to hardware threads, but having more application threads than physical threads opens opportunities to address asynchrony. This is commonly known as "overdecomposition", and if used opportunistically can avoid blocking of hardware resources. It is done by switching out an application thread that has been blocked while waiting for a long-duration access or service and putting a pending thread on the hardware to continue using these resource, improving efficiency and scaling. This requires the ability to do on-the-fly context switching. As the granularity of threads can be made finer and still be efficient, this permits an increase in scalability for strong scaling and more threads with weak scaling. Hardware for multithreaded execution has been developed, including the Tera MTA with 128 threads with single-cycle context switching times.

21.5.2 MESSAGE-DRIVEN COMPUTATION

The historical load/store architecture combined with message passing for scalability has proved effective over more than 25 years. But with increasing scale and resulting asynchrony, the effects of latency have proven increasingly costly in dimensions of both time and energy. Where both temporal and spatial locality could be exploited by caches, the deleterious effects of latency could be bounded, especially with cache-aware programming techniques. However, more general computation, extended scale, and broader access patterns expose system latencies and uncertainties. An alternative stratagem is message-driven computations that move work to the data rather than always demanding that data is moved to the nexus of the static work control state. Combined with multithreading, this technique can hide some latencies, especially long-distance latencies, and avoid blocking hardware as a result. By keeping the work and the data close together, it can reduce absolute latencies of action.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 597

Within computing research there is a long heritage of exploration of message-driven computation. Dataflow architectures of the 1970s and 1980s considered lightweight messages called "tokens" to move intermediate result data between succeeding actors, referred to as "templates". The actors' model added semantic richness to this in the form of "futures". In 1992 Dally's J-machine provided hardware support for messages to instantiate methods for remote procedure calls. The University of California at Berkeley, as part of the threaded abstract machine model, devised a version of message-driven computing called "active messages", a term that has had some traction. More recently the ParalleX execution model and the HPX runtime systems it inspired incorporate "parcels" that convey actions to be performed on remote data, as well as the means to support the migration of continuations to provide for dynamic placement of the parallel control state.

In essentially every case, a lightweight message is structured to incorporate several fields of information. These include a destination, an action specification, a payload field, and in some cases a continuation. The destination can be a physical node, software process, core thread, or virtual data object. The action can be as simple as an operation, a sequence of instructions, a pointer to a method or procedure, or an effect on some element of control state or synchronization object. The payload varies from nothing or void to a set of independent scalars, vector, list, or a more general structure. These values (or pointers) are used along with the destination object data to perform the projected calculations. The final field is referred to as the "continuation", which in simplest terms tells the destination what to do after the specified action is completed. This can be as simple as return the result to the originating source of the action—typical of more conventional computing. It also can indicate what recovery response to an error should occur. But more interesting in some models is the effect on the global control state either by modifying the state of an existing control object or adding such a control object to the existing global parallel control state.

21.5.3 GLOBAL ADDRESS SPACE

A division in thinking about scalable computing has existed for more than two decades concerning support for global address spaces. This single issue has incited severe argument on occasion, as substantial investment has been made in both classes of system. In truth, it is much harder to design a very large computer of many, perhaps thousands, of nodes that retains uniformity of address spaces across the entire system. While delivering a means to access known physical addresses correctly is achievable, the more challenging problems are dealing with virtual addresses and cache coherency. One approach employed for virtual address is the partitioned global address space (PGAS). Here the virtual address space is partitioned as contiguous blocks among the physical nodes. The upper bits of the virtual address identify the node on which it will be found. This is efficient, but it has the unfortunate property that a word associated with a given virtual address cannot be moved between nodes and retain the same virtual address. Cache coherence is even more challenging, as any processor becoming the owner of a virtual location, i.e., it can write to it, must be able to invalidate all copies of that location throughout the distributed system. In some cases cache coherency is not assumed and remote accesses are differentiated from local accesses, only the latter of which within a given node is treated as cache coherent. Yet another problem is implicit: locality and latency. A strong argument for

message passing in a distributed-memory address space such as that used with MPI is that it forces the programmer to deal explicitly with locality, optimizing local operations and minimizing global accesses through message passing. This has proven quite effective for many applications. Nonetheless, the inefficiencies of using message passing for lightweight remote data accesses and the increased difficulties in having to control data movement explicitly by this means has led to many experimental AMT software systems incorporating global address space frameworks, at least of the PGAS type.

21.5.4 ACTOR SYNCHRONIZATION

Conventional programming methods, particularly of the message-passing forms but also for multiple-thread operation, use global barriers, blocking message send/receives, or nonblocking send/receives with waiting. These are semantically weak, in that they accomplish relatively little in flow control while incurring significant overheads. They tend to be coarse grained, especially in the case of global barriers where all processes or threads must come to a stop until all tasks have completed their respective work prior to the barrier synchronization point. Some AMT systems incorporate advanced dynamic synchronization constructs such as dataflow and futures, both of which have a heritage extending back more than 3 decades. Dataflow addresses out-of-order completion of input operands and asynchrony of arrival prior to scheduling a specified operation to be performed. The futures synchronization extends this to different uses of the same prior result value requested by other streams of execution delivering the equivalent of an IOU that can be treated as a manipulatable pointer to an eventual value and employed in building data structures giving additional parallelism.

21.5.5 RUNTIME SYSTEM SOFTWARE

The concepts described in the subsection on AMT are found most readily in a small number of runtime software libraries developed for scalable and efficient HPC. Work on such runtimes as Charm++ from the University of Illinois Urbana—Champaign, OCR from Rice University, and HPX from Louisiana State University and Indiana University is representative, but these are by no means the only packages. They vary in detail, sometimes in important ways, but have many similarities in their main functionality and semantics. Runtime software is the easiest way to achieve dynamic adaptive computing, and for some classes of applications such as adaptive mesh refinement, molecular dynamics, particle in cell, fast multipole methods, and dynamic graph problems including data analytics it can serve well for improved efficiency, scalability, and user productivity. But for some applications there is little or no performance improvement, in part because runtime software actually adds to the total overhead imposed on the system. Some cases have been documented where performance actually degrades with scale for this reason. Runtime software can manage over-decomposition and this often makes better utilization of computing resources, at least up to a point. But runtime software behavior is also sensitive to scheduling policies, which may need to adapt to application algorithm requirements. In the future it is hoped that hardware architectures will evolve to incorporate mechanisms for accelerating certain aspects of runtime system operation to lower overhead and improve useful parallelism.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 599

21.6 THE NEODIGITAL AGE

After decades of exponential growth of semiconductor technology, Moore's law is coming to an end, if it has not already done so (depending on how its defined). This explosion of on-chip components can no longer be relied upon to deliver continued performance gain. Even over the last decade clock rates have flattened due to power limitations, and instruction-level parallelism has also flatlined despite optimistic expectations. This period has seen performance delivery gains achieved principally through multicore and many-core processors using up the last vestiges of Moore's law as enabling technology reaches nanoscale. If Moore's law can no longer yield greater performance given the other limitations, what can?

Here we hypothesize a new generation of HPC systems that differ significantly from conventional practices and their incremental extensions. Design strategies include the following:

- *Execution models.* The history of high-end computing has experienced about half-a-dozen paradigm shifts over the last 7 decades to adopt new enabling technologies and exploit different forms of hardware parallelism. Such phase changes include the original von Neumann architecture, vector and SIMD processing, and communicating sequential processes and shared-memory multithreaded. But new execution models are required that dramatically increase both scalability and efficiency to drive future computing across the exascale performance region, even possibly to zetaflops.
- *Architecture fundamentals.* Originally, floating-point operations were the performance-limiting property of HPC systems and the early architectures were designed to achieve highest arithmetic unit utilization. Efficiency is often described as the ratio of sustained floating-point performance to peak floating-point performance. But today memory bandwidth is the critical resource and memory capacity the biggest single cost factor. Overall data movement system-wide is also time limiting. Future architectures need to be redesigned around these performance and energy boundaries rather than historical biases.
- *Parallel algorithms.* How we organize a computation is highly sensitive to the nature of the machine structure upon which a problem is to be executed. Changes in algorithms are required to adjust to new structures and expose and exploit parallelism intrinsic to the target problem domain, as well as effective memory usage. There have been many instances when algorithmic changes have dramatically improved overall computational time to solution.
- *Programming interfaces.* The semantics of control and data are reflected by the programming interfaces, including languages and libraries that determine the means of applying applications to HPC systems. As system architectures have evolved over the decades, changing the forms of parallelism they exploit, programming methods have to evolve as well to let programmers take advantage of the hardware. We have watched this as MPI has evolved from MPI-1 to MPI-4. Programming interfaces are often extended, and sometimes new ones are created. The creation and use of CUDA allow programmers to take advantage of the peculiarities and opportunities made possible by the architecture improvements.

In the following sections examples of approaches to computing beyond conventional practices are presented to hint at possible elements of future HPC systems and methods. Some of these ideas have a long legacy in the research community. Others have yet to be investigated in any depth but are considered (by the authors at least) to have future promise.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 600

21.6.1 DATAFLOW

Dataflow is a parallel execution model originally developed in the 1970s but explored and enhanced as the basis for non-von Neumann computing architecture and techniques. While there were many contributors over a period of more than two decades, two investigators stand out: Jack Dennis and Arvind, both at Massachusetts Institute of Technology. These two leaders in the field, although at the same institution, led separate research groups and pursued distinct conceptual strategies. Jack Dennis, who may be considered the father of dataflow, founded what has now come to be called "static dataflow", while Arvind is credited with the introduction of the school of "dynamic dataflow". Strong research programs, especially in the 1980s, were conducted worldwide, with full implementations of hardware systems in the United States, Japan, and Europe. Some fundamental flaws in architecture reflecting the abstract model naively ultimately doomed this particular approach, largely due to overheads. But the underlying concepts are important and contribute to many hardware and software techniques, although not in the original forms anticipated by its founders. It is quite likely that innovative approaches exploiting the valuable aspects of dataflow will drive future system architectures and programming methods at the end of Moore's law in nanoscale fabrication technology.

Dataflow in its purest form represents a computation in terms of the data precedence constraints of the operations to be performed. A visual presentation of a dataflow program looks like a directed graph, with its vertices determining the operations to be performed and the links between vertices determining the flow of operand values, from the source vertices that produce the intermediate values of the computation to the destination vertices that consume the values as input operands to their own operation. "Tokens" were initially expected to serve as messages that carry these values from the output of the source where the values are calculated to the input of the destinations where the values are used for follow-on calculations. The operations themselves are designated by a small data structure or record referred to as a "template", which specifies the operation to be performed, buffers the input values, designates the destination templates for the result values, records and updates synchronization control state, and includes other information as required by the specific design.

Dataflow is a functional or value-oriented model of computation. There is no shared data; no global side-effects. Only actual values can be exchanged between operator templates. This has many advantages, at least in the abstract. In its original version dataflow was fine grained, revealing most of the intrinsic parallelism. It is very robust, avoiding many of the pitfalls associated with shared-memory models. As intrinsics, tokens provide event-driven computing through self-synchronization and templates maintain control state, permitting operations to be performed only when all operand values have been received, although order of arrival does not matter. Problems with aliasing and race conditions are thus avoided. Backus (inventor of Fortran) in his famous Turing Award lecture strongly advocated functional programming as the only means of writing robust code. A number of functional programming languages have been developed over the decades, with Haskell the most recently and widely used example.

Dataflow, at least as manifest, suffered from a number of inefficiencies which ultimately made it nonviable as a basis for hardware architecture. Perhaps most egregious was that it was inefficient due to overheads of operation control and scheduling. Many microoperations were required for each template operation performed, yet those operations were very lightweight and did not amount to a lot of work. This meant that more work was performed in managing a template than the resulting full operation performed. Compared to a typical program counter, this was far more effort. It was sensitive

to bubbles or gaps in the operational pipeline because of the need for all these update events for each single useful operation. It was memory intensive due to requirements of template storage. Perhaps worse was that it was also memory bandwidth intensive, again due to all the data transfer and synchronization events. This was compounded by the dissemination of result values to possibly many destination templates. The failure to take advantage of usual accelerating mechanisms such as registers, caches, and reservation stations made it difficult for it to compete effectively with RISC uniprocessor architecture, although these were sequential and used execution pipelines with out-of-order completion, effective compiler methods, higher clock rates, and lower energy. Finally, in a period when parallel computers were limited to at most a few hundred processors, and this rare, coarse-grained parallelism was sufficient to keep the resources fully utilized. Full use of the fine-grained operation implied by dataflow was unnecessary and in fact wasteful.

In spite of these deficiencies of the original dataflow architecture designs, the underlying execution model is very powerful. It addresses the key challenge of asynchrony and the resulting uncertainty of order of operational events. It makes for cleaner composability of separately developed software. It provides a clean means of overdecomposition which can be used for dynamic adaptive resource management and task scheduling to avoid resource blocking and circumventing contention over shared resources. It provides a natural way to minimize starvation by using much of the available parallelism. There are many possible variations, compromises, and hybrid structures that may be able to benefit from the dataflow concept. Already there is increased use of directed acyclic graph representation of computations at the medium- or coarse-grained level to exploit more adaptive flow control for enhanced efficiency and scalability. For these reasons it is projected that where new architectures will be required to increase performance at the end of Moore's law, concepts embodied by the dataflow execution model will be employed, although in innovative ways.

21.6.2 CELLULAR AUTOMATA

Among the many contributions by the mathematician John von Neumann was the invention of a distinct model of computation, cellular automata, in 1949. The irony cannot be avoided: the cellular automaton is considered a "non-von Neumann architecture".

In its simple form, a cellular automaton consists of a two-dimensional array (it can be one- or three-dimensional as well) of cells, each of which is connected to its nearest neighbors (typically four: up, down, right, left). Each cell contains a small amount of state: sometimes a few bits or a few words, although it can be more. Finally, a cell incorporates a set of rules that determines how its own state will change depending on its current state and that of its immediate neighbors.

The classic example of cellular automata is Conway's Game of Life, in which each cell has one of two states (e.g., alive or dead), and a small set of simple rules as follows.

- A cell dies if it is alive and only one or none of its neighbors is alive.
- If two or three of the neighbors are live, then an alive cell remains in that state.
- But if an alive cell has four, five, or six live neighbors, then it dies.
- Finally, a dead cell with three live neighbors becomes live.

The evolution of this cellular automaton is determined entirely by its initial state, that is the state of all its cells. Von Neumann was able to demonstrate that there is a set of rules and state that is Turing equivalent and therefore, in principle, can serve as a general-purpose computer, although his solution

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 602

was theoretical and did not reflect a practical framework for real-world computing. Conway's model is much simpler, and it too is a universal Turing machine.

A cellular automaton has a number of properties that makes it interesting as a future class of HPC architecture. It exposes an enormous amount of hardware parallelism, as each cell can be very small and thus for a given fabrication feature size and die area there can be a maximum number of execution units, although of modest capabilities. It has very large storage bandwidth, although the memory density may not be as good as other means. Latency for local storage access will be very low, as well as to the nearest neighbor state. If the communication is flat as described, remote access could take many hops and impose long latencies. However, hierarchical topologies can strongly mitigate this. Asynchrony across the very wide system can also be ameliorated by local synchronization built into the hardware functionality. A number of data and operation sequence layout patterns can be devised to take advantage of vector, systolic, SIMD, wave, dataflow, and graph algorithms derived from prior art.

There are many open questions, many tradeoffs in balancing storage capacity, operation functions, and communication, and the major challenge of achieving the global emergent behavior of general-purpose parallel computing from massive local basic operations across the array of cells. Essentially, what is the new execution model? It is not clear that this is ultimately possible. However, it does open a new class of HPC architecture at a time when architecture may be the only hope of a significant performance advance.

21.6.3 NEUROMORPHIC

The human brain is an extraordinary system, perhaps the most complex known. It incorporates more interconnections within a single cranium than all the stars in the Milky Way galaxy. With 89 billion neurons it consumes only 20 W of power, with each neuron on average communicating through 10,000 synaptic junctions. While each neuron operates at less than 1 kHz, a brain activates more than 10 trillion spikes per second. And it is constantly changing in topology to fix long-term memories. Certain forms of operation, such as associative processing of images, sounds, and patterns, are accomplished with a throughput unachievable by even the highest-performance supercomputer. Then, speculate researchers, cannot a future class of computers be developed around the same principles as the human brain to make artificial computing systems with similar remarkable capabilities? In the most general sense these are known as "neuromorphic" computers; there are many diverse approaches being explored at this time, but all are inspired by the brain.

21.6.4 QUANTUM COMPUTING

Trying to explain quantum computing is like teaching computer science at Hogwarts. But quantum computing is not magic, even if it seems like it. And in no practical sense is it real yet. But it is both theoretically possible and becoming ever more likely technically, although there is still some way to go. Why there is so much interest and investment in what is still a research domain is the capability and impact that it would have if actually achieved. Again theoretically, it would be possible to do certain calculations that would be impossible with conventional supercomputers even if running for the entire age of the planet. And this is not limited to some esoteric or obscure problem, but rather for areas of extreme criticality such as cryptanalytics or multidimensional nonlinear optimization.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 603

The fundamental concepts behind quantum computing have been understood since the 1980s, including the foundational work by the Nobel Laureate Richard Feynman of Caltech, among other pioneers. The core idea for quantum computing is embodied in the idea of the "qubit" that stores quantum information as the Shannon bit stores binary information. But there the similarity ends. A qubit state is not a 0 or a 1 but rather the superposition of the probabilities of the data being either a 0 or a 1. A set of n qubits can store the probabilistic distribution of all possible values, that is 2^n possible values, and process all of these simultaneously, at least in principle. The sum of the probabilities must be equal to 1. When the values of the hypothetical qubit are measured (observed), the output value collapses to a single n-bit answer. The likelihood of any particular answer is the probability of the superimposed field associated with that value. That means that rerunning the computation a number of times is likely to deliver different values.

Important breakthroughs came with the development of particular algorithms that showed how theoretical quantum computers could be employed to accelerate certain problems. Shor's algorithm for factorization was an example that spawned significant interest and research in this field. A variant of such machines, called "quantum annealing" computers, perform a narrower range of optimization algorithms. In spite of its limitations, practical systems of this type have been built and operated, in particular by the Canadian company D-wave.

The technology required to achieve this functionality, at least as understood through actual experimentation, involves extremes in cryogenic superconductivity. Specifically, Josephson Junctions cooled to 10 s of millikelvins (degrees Celsius above absolute zero) are employed to maintain stability of the quantum state long enough to perform the required computation. It is not easy, and alternative methods are under research. Whether or not a viable technology solution is ultimately developed, the advantages of full quantum computing are still limited. There are certainly many classes of problems performed by conventional computers today that cannot be accelerated by a quantum computer as currently conceived. Also, it is not possible that a future quantum computer can in principle solve a problem not solvable by conventional computers, in that they are Turing equivalent. Nonetheless, for those problems with a conceivable performance advantage, the prospects for futuristic quantum computers are very exciting.

21.7 EXERCISES

1. Look up the most energy-efficient supercomputer on the Green 500 list [3]. Extrapolate the top system to exascale and estimate the power cost to operate. How close is it to the goal of 20 MW/year for an exascale machine?
2. Review the most recent Gordon Bell Prize winners [4]. How many applications used "MPI+X"? How many utilized GPUs? What architectures are represented?
3. What kinds of applications will benefit from exascale computing resources? What kinds of applications will not benefit from such resources?
4. What types of applications are currently being deployed on commercial quantum computers?
5. Explain why Moore's law has given a free ride to application developers in improving application performance. What are the consequences of Moore's law ending?

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 604

REFERENCES

[1] T. Sterling, Personal Communication.
[2] Wikipedia, Larrabee (Microarchitecture), [Online]. https://en.wikipedia.org/wiki/Larrabee_(microarchitecture).
[3] Green500, Green500 List, [Online]. https://www.top500.org/green500/.
[4] Association for Computing Machinery, ACM Gordon Bell Prize, [Online]. http://awards.acm.org/bell/.

Essential C

This appendix is intended to assist those who already know one or more programming languages and may need a brief introduction to C syntax and semantics. The approach provided here is driven by four examples which address a wide range of C usage: numerical integration, lower/upper (LU) decomposition, the fast Fourier transform (FFT), and the game of tic-tac-toe. In these examples the core C syntax is illustrated. Some of the key C syntax elements are highlighted in Table A.1.

A.1 NUMERICAL INTEGRATION

Integrating an ordinary differential equation using a method from the Runge–Kutta family of integrators is a common task in scientific computing. The classic fourth-order algorithm, frequently referred to as rk4, illustrates many important features of the C programming language. In the example presented in Code 1, which solves the ordinary differential equation

$$\frac{dx}{dt} = -\lambda x$$

for the function $x(t)$, there are only two routines required: *main* and *rhs*. While the *main* routine is present in all C codes, the *rhs* function provides the right-hand side evaluation of the ordinary differential equation evaluated at different function values and times. The header for the *rhs* function is declared in line 13, while the function itself is declared in lines 71–73. The header provides *main* with information on the expected input and output of the function and aids in type checking. Output from each step of the Runge–Kutta integration is written to a file. The file handler is declared in line 35, while the output file itself, *rk4.dat*, is opened in line 38 with "write" access, as indicated by the *w* in the last argument to the routine *fopen*. Output from each step of the integration is written to the file using the *fprintf* function in line 54, and the file is closed in line 65 using *fclose*. The rk4 integrator itself is listed in lines 56–63. The exact solution to this ordinary differential equation is

$$x(t) = Ce^{-\lambda t}$$

where the constant $C = 1$ based on the initial condition for $x(t)$ given in line 41. The exact solution is evaluated in line 49 and used to evaluate the error in the numerical solution throughout the integration.

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.15001-4
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 606

Table A.1 Some Key C Syntax Elements Used in the Four Example Codes Here

Operational or Functional Element	Location	Example
For loop	Code 1, line 44	`for (i=0;i<N;i++) {`
While loop	Code 3, line 11, 85	`while (nmoves++ < 9) {`
If statement	Code 4, line 90	`if (!p->level) return;`
Bitwise right shift	Code 3, line 12	`k >>= 1;`
Bitwise left shift	Code 3, lines 27,28	`if(n & (1 << (log2_int(N) - j)))`
Bitwise AND	Code 3, lines 20, 27,58	`if(!(i & n)) {`
Bitwise OR	Code 3, line 28	`p |= 1 << (j - 1);`
Modulo	Code 3, line 60	`double complex Temp = W[(i * a) % (n * a)] * f[i + n];`
Increment operator	Code 1 line 44 Code 4 line 148	`for (i=0;i<N;i++) {`
Subtract and assignment operator	Code 2, lines 71,74	`for (int j = 0; j < i; j++) x[i] -= ap[i][j]*x[j];`
Conditional Expression	Code 4 line 55	`char who = (upper->who == comp_mark)? user_mark: comp_mark;`
Structures	Code 4 line 10	`typedef struct move {`
Allocate and zero initialize	Code 4 line 60	`if (!m) m = calloc(1, sizeof(move_t));`
Complex allocation	Code 3, line 46	`W = (double complex *)malloc(N / 2 * sizeof(double complex));`
Powers of complex numbers	Code 3, line 52	`W[i] = cpow(W[1], i);`
File handling	Code 1, lines 38,54,65 Code 2, lines 19, 33,34	`FILE *f = fopen(path, "r");`
Getline	Code 2, line 23	`getline(&line, &n, f);`

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 607

```
0001 /*
0002 * Solving the Ordinary Differential Equation
0003 *  dx/dt = -lambda x
0004 *  using Runge-Kutta 4th order
0005 */
0006
0007 #include <stdio.h>
0008 #include <stdlib.h>
0009 #include <math.h>
0010
0011 #define lambda 5.0
0012
0013 double rhs(double x,double t);
0014
0015 int main(int argc, char *argv[]) {
0016   // Starting and Stopping integration time
0017   double A = 0.0;
0018   double B =  1.0;
0019
0020  // Size of timestep
0021  double dt = 0.1;
0022
0023  // Number of timesteps
0024  int N = (B-A)/dt+1;
0025
0026  int i;
0027
0028  // predictor-corrector rk4 steps
0029  double F1,F2,F3,F4;
0030  double x,t;
0031  double exact;
0032  double small_number = 1.e-15;
0033
0034  // file for output
0035  FILE *rk4data;
0036
0037  /* open a file for output */
0038  rk4data = fopen("rk4.dat","w");
0039
0040  /* initial condition: x(A) = 1.0 */
0041  x = 1.0;
0042
0043  fprintf(rk4data,"# t x(t) exact diff\n");
0044  for (i=0;i<N;i++) {
0045  /* RK4 */
0046  t = A + dt*i;
0047
0048  /* Exact solution */
0049  exact = exp(-lambda*t);
```

```
0050
0051  /* Write to file the log of the difference
0052  * between the exact solution and integrated solution */
0053  //fprintf(rk4data,"%g %g\n",t,log(x-exact+small_number));
0054  fprintf(rk4data,"%g %g %g %g\n",t,x,exact,x-exact);
0055
0056  // Runge-Kutta
0057  F1 = rhs(x,t);
0058  F2 = rhs(x+0.5*dt*F1,t+0.5*dt);
0059  F3 = rhs(x+0.5*dt*F2,t+0.5*dt);
0060  F4 = rhs(x+dt*F3,t+dt);
0061
0062  // update x(t)
0063  x = x + 1.0/6.0*dt*( F1 + 2.0*F2 + 2.0*F3 + F4 );
0064  }
0065  fclose(rk4data);
0066
0067  return 0;
0068  }
0069
0070  // The right hand side of the first order differential equation
0071  double rhs(double x,double t) {
0072  return -lambda*x;
0073  }
```

Code 1. Integrating an ordinary differential equation using a Runge–Kutta method in C.

A.2 LOWER/UPPER DECOMPOSITION

The LU decomposition program is an example of dense algebra processing. Such problems deal with matrix and vector representation of systems of equations, potentially including thousands of interrelated equations and variables. Among many other applications, matrix-based equation solvers are used to generate numerical solutions of ordinary and partial differential equations, and thus are widely employed in simulations of real-world phenomena in nearly every branch of physics.

The program, listed in Code 2, solves the equation $Ax = b$ for x. Matrix A and vector b are given to and retrieved from files passed to the program as the required command-line arguments. To be able to solve the system, A must be square and contain linearly independent columns and rows. To calculate the solution taking into account a broad range of possible instances of A, the code employs a specific approach in which the matrix is expressed as a product of a lower triangular matrix L (all its elements above the main diagonal are zero) and an upper triangular matrix U (all elements below the main diagonal are zero). Since converting the original matrix to triangular form requires that the so-called pivot elements located on the main diagonal of A are nonzero (the algorithm divides by pivot value to compute values of other elements of the matrix), on occasion rows of the matrix must be swapped to avoid this issue. Note that this simply corresponds to changing the order in which original equations are listed, and therefore does not change the solution. The reordering is expressed by a permutation

matrix P, which has a single 1 in every column and row and 0s elsewhere. The matrix is expressed thus as:

$$PA = LU$$

Leading to the problem definition as

$$LUx = Pb$$

This may be further decomposed into two related systems by introducing an intermediate vector y:

$$Ly = Pb$$

$$Ux = y$$

Thanks to the fact that L is triangular, obtaining the solution of the first equation may be done trivially via forward substitution. Note that the first element of y may be calculated by dividing the first element of permuted vector b by the element in row 1 and column 1 of L. Since the second row of L contains only two nonzero elements, the second element of y may be computed based on the value of the first element. This process continues row by row until all elements of y are known. The calculated y can be plugged into the second matrix equation involving U, and used to compute the values of vector x using an analogous approach (back substitution).

The contents of matrix A and vector b are fetched from storage using various file I/O operations seen in the *read_from_file* function (*fopen*, *getline*, *fread*, and *fclose*). The first of them (invoked in line 19) opens a file described by *path* for reading. The file contains a header with one or two decimal numbers separated by a space and terminated with a new-line character; they describe the dimensions of the matrix or vector. This part is extracted using the *getline* function (line 23) that stops after reading the full line, followed by the sscanf function (line 24) that converts the text representation of the numbers into integer variables. Since *sscanf* returns the number of items converted, the following switch statement explicitly zeroes out each dimension that could not be read. The routine allocates the storage for the array or vector and treats the remaining contents of the file as serialized binary data (a sequence of double-precision floating-point numbers without gaps). The data is stored in row-major form, starting with the lowest row index and the lowest column index in each row. While this approach results in efficient use of storage space, the files are not portable to architectures with incompatible memory layout, i.e., different endianness and different number size. Also, due to difficulties associated with inspecting the contents of such files, the use of portable and self-describing file formats such as HDF5 or NetCDF is recommended. The transfer of the file contents to data memory is performed by the *fread* function in line 33. After the number of retrieved elements (not bytes!) is verified, the file is closed by the *fclose* function (line 34).

The LU decomposition is performed by the *decompose* function. To conserve memory, a commonly used trick stores both L and U matrices in the space occupied originally by A. This is possible because matrix L has only ones on its diagonal, thus eliminating the need for their explicit storage, so the diagonal elements of A can be used to keep the corresponding elements of matrix U. To accelerate the computations and enable more familiar double-index notation when accessing the matrix elements, an array of pointers to rows *ap* is allocated and initialized. This allows for much faster row permutations. To swap two rows, only their pointers are exchanged; the contents of rows are kept in their original memory locations. The decomposition scans the values of elements underneath each

diagonal component to select the largest (in absolute value sense) pivot in each row. This is both to avoid zero- or near zero-valued pivots and to obtain good numerical accuracy. Since the permutation matrix P is very sparse (recall that it has only one nonzero element per row), a permutation vector is used instead. It stores the updated index for each row that will be later used to fetch the elements of vector b correctly. Originally each index in P points to the initial position of the row; whenever the rows are swapped, the corresponding entries in P are swapped as well.

The *solve* function computes the solution vector x in two steps. The first calculates the intermediate vector from L and vector b accessed via the redirection vector p. The seconds step converts the elements of the intermediate vector into final solution values. No extra storage is needed, since the elements of the intermediate vector are consumed at the same rate as the elements of x are produced.

If the user specifies the third argument on the command line, it will be interpreted as a path name to the file storing the solution vector. Function *write_to_file* handles the data output using similar I/O functions as those used for reading, but with the exception of *fprintf* (line 84) and *fwrite* (line 85). The first behaves much like the regular *printf*, except it redirects its output to the stream specified as its first parameter. The *fwrite*, on the other hand, accepts the same arguments as *fread* and writes the raw data to the opened file. If the third argument is not given, the program prints out all elements of vector x (lines 106—108).

The program uses the *variadic* function (that is, a function accepting a variable number of arguments) *error* to handle critical execution problems. This is indicated by the ellipsis ("...") as the last formal parameter. Thanks to variadic support, the programmer may pass additional information to be included in the error messages. The definitions required to access this functionality are provided by the include file "stdarg.h".

```
0001 #include <stdio.h>
0002 #include <stdlib.h>
0003 #include <stdarg.h>
0004 #include <math.h>
0005
0006 #define EPS 1e-20
0007
0008 // handle error occurrence
0009 void error(char *fmt, ...) {
0010   va_list ap;
0011   va_start(ap, fmt);
0012   fprintf(stderr, "Error: ");
0013   vfprintf(stderr, fmt, ap);
0014   exit(1);
0015 }
0016
0017 // read in contents of matrix or vector from file
0018 double *read_from_file(char *path, unsigned dim[2]) {
0019   FILE *f = fopen(path, "r");
0020   if (!f) error("cannot open \"%s\" for reading\n", path);
0021   size_t n = 0;
0022   char *line = NULL;
0023   getline(&line, &n, f);
```

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 611

```
0024   switch (sscanf(line, "%u %u", &dim[0], &dim[1])) {
0025     case 0: dim[0] = 0;
0026     case 1: dim[1] = 0;
0027       break;
0028   }
0029   if (!dim[0]) error("invalid data file format in file \"%s\"", path);
0030
0031   n = dim[0]*(dim[1]? dim[1]: 1);
0032   double *data = malloc(sizeof(double)*n);
0033   size_t cnt = fread(data, sizeof(double), n, f);
0034   fclose(f);
0035   if (cnt < n) error("file \"%s\" seems to be truncated\n", path);
0036   return data;
0037 }
0038
0039 // perform LU decomposition
0040 double **decompose(int n, double *a, double *b, int *p) {
0041   double **ap = malloc(sizeof(double)*n); // array of row pointers
0042   for (int i = 0; i < n; i++) {
0043     ap[i] = &a[i*n]; p[i] = i;
0044  }
0045
0046  for (int i = 0; i < n; i++) {
0047    for (int j = i+1; j < n; j++) {
0048      int maxind = i;
0049      if (fabs(ap[j][i]) > fabs(ap[maxind][i])) maxind = i;
0050      if (maxind != i) {
0051        double *atmp = ap[i];
0052        ap[i] = ap[j]; ap[j] = atmp; // pivot row swap
0053        int ptmp = p[i];
0054        p[i] = p[j]; p[j] = ptmp; // permutation tracking
0055      }
0056      if (fabs(ap[i][i]) < EPS) error("matrix A ill-defined, aborting\n");
0057   }
0058   for (int j = i+1; j < n; j++) {
0059     ap[j][i] /= ap[i][i];
0060     for (int k = i+1; k < n; k++) ap[j][k] -= ap[j][i]*ap[i][k];
0061     }
0062   }
0063   return ap;
0064 }
0065
0066 // solve system of equations
0067 double *solve(int n, double **ap, double *b, int *p) {
0068   double *x = malloc(sizeof(double)*n);
0069   for (int i = 0; i < n; i++){ // forward substitution with L
0070     x[i] = b[p[i]];
0071     for (int j = 0; j < i; j++) x[i] -= ap[i][j]*x[j];
0072   }
```

```
0073   for (int i = n-1; i >= 0; i--) { // backward substitution with U
0074     for (int j = i+1; j < n; j++) x[i] -= ap[i][j]*x[j];
0075     x[i] = x[i]/ap[i][i];
0076   }
0077   return x;
0078 }
0079
0080 // save result vector to file
0081 void save_to_file(char *path, int n, double *x) {
0082   FILE *f = fopen(path, "w");
0083   if (!f) error("cannot open \"%s\" for writing\n", path);
0084   fprintf(f, "%d\n", n);
0085   if (fwrite(x, sizeof(double), n, f) != n)
0086     error("short write to \"%s\" file\n", path);
0087   fclose(f);
0088 }
0089
0090 int main(int argc, char **argv) {
0091   // initialization and sanity checks
0092   if (argc != 3 && argc != 4)
0093     error("usage: %s matrix_file vector_file [result_file]\n", argv[0]);
0094   int Asize[2], bsize[2];
0095   double *A = read_from_file(argv[1], Asize);
0096   if (Asize[0] != Asize[1]) error("matrix A is not square\n");
0097   double *b = read_from_file(argv[2], bsize);
0098   if (bsize[1] > 0) error("b is not vector\n");
0099   if (bsize[0] != Asize[0]) error("size of b incongruent with A\n");
0100   int *P = malloc(sizeof(int)*(*bsize)); // row permutation vector
0101   // decompose and solve
0102   double **Ap = decompose(*bsize, A, b, P);
0103   double *x = solve(*bsize, Ap, b, P);
0104   // output handling
0105   if (argc > 3) save_to_file(argv[3], *bsize, x);
0106   else { // if no output file specified, print out the solution
0107     for (int i = 0; i < *bsize; i++) printf("%f ", x[i]);
0108     printf("\n");
0109   }
0110
0111   return 0;
0112 }
```

Code 2. Source code for matrix LU decomposition and solver.

A.3 FAST FOURIER TRANSFORM

The FFT is a core computational science algorithm that usually involves complex numbers. Code 3 illustrates computing the FFT in C using the Cooley–Tukey algorithm. This implementation illustrates

the use of the computing powers of complex numbers (line 52), bit right-shift assignment (line 12), the bitwise AND operator (lines 20, 27, and 58), bitwise OR assignment (line 28), bitwise left shift (lines 27 and 28), modulo (line 60), while-loops (lines 11 and 85), for-loops, and if conditionals. It is an example of a divide-and-conquer algorithm.

```
0001 #include <stdio.h>
0002 #include <stdlib.h>
0003 #include <math.h>
0004 #include <complex.h>
0005
0006 #define MAX 200
0007
0008 int log2_int(int N)    /*function to calculate the log2(.) of int numbers*/
0009 {
0010    int k = N, i = 0;
0011    while(k) {
0012      k >>= 1;
0013      i++;
0014    }
0015    return i - 1;
0016 }
0017
0018 int check(int n)    //checking if the number of element is a power of 2
0019 {
0020    return n > 0 && (n & (n - 1)) == 0;
0021 }
0022
0023 int reverse(int N, int n)    //calculating revers number
0024 {
0025   int j, p = 0;
0026   for(j = 1; j <= log2_int(N); j++) {
0027     if(n & (1 << (log2_int(N) - j)))
0028       p |= 1 << (j - 1);
0029   }
0030   return p;
0031 }
0032
0033 void ordina(double complex* f1, int N) //using the reverse order in the array
0034 {
0035   double complex f2[MAX];
0036   for(int i = 0; i < N; i++)
0037     f2[i] = f1[reverse(N, i)];
0038   for(int j = 0; j < N; j++)
0039   f1[j] = f2[j];
0040 }
0041
0042 void transform(double complex* f, int N) //
```

```
0043 {
0044   ordina(f, N); //first: reverse order
0045   double complex *W;
0046   W = (double complex *)malloc(N / 2 * sizeof(double complex));
0047   double rho = 1.0;
0048   double theta = -2. * M_PI / N;
0049   W[1] = rho*cos(theta) + rho*sin(theta)*I;
0050   W[0] = 1;
0051   for(int i = 2; i < N / 2; i++) {
0052     W[i] = cpow(W[1], i);
0053   }
0054   int n = 1;
0055   int a = N / 2;
0056   for(int j = 0; j < log2_int(N); j++) {
0057     for(int i = 0; i < N; i++) {
0058       if(!(i & n)) {
0059          double complex temp = f[i];
0060         double complex Temp = W[(i * a) % (n * a)] * f[i + n];
0061         f[i] = temp + Temp;
0062         f[i + n] = temp - Temp;
0063        }
0064      }
0065      n *= 2;
0066      a = a / 2;
0067   }
0068 }
0069
0070 void FFT(double complex * f, int N, double d)
0071 {
0072   transform(f, N);
0073   for(int i = 0; i < N; i++)
0074     f[i] *= d; //multiplying by step
0075 }
0076
0077 int main()
0078 {
0079   int n;
0080   do {
0081     printf(" Give array dimension (needs to be a power of 2)\n");
0082     char str1[20];
0083     scanf("%s",str1);
0084     n = atoi(str1);
0085   } while(!check(n));
0086   double sampling_step = 1.0;
0087   double complex vec[MAX];
0088   double freq = 100;
0089   double x;
0090   printf(" Input vector\n");
```

```
0091    for(int i = 0; i < n; i++) {
0092        x = -M_PI + i*2*M_PI/(n-1);
0093        vec[i] = cos(-2*M_PI*freq*x);
0094        printf("%g + %g I\n",creal(vec[i]),cimag(vec[i]));
0095    }
0096    printf("-------------------\n");
0097    FFT(vec, n, sampling_step);
0098    printf(" FFT of the array\n");
0099    for(int j = 0; j < n; j++)
0100      printf("%g + %g I\n",creal(vec[j]),cimag(vec[j]));
0101
0102     return 0;
0103 }
```

Code 3. FFT in C. This code was adapted from a C++ version in Wikipedia [1].

A.4 GAME OF TIC-TAC-TOE

To illustrate a problem with dynamically generated and deleted data structures, Code 4 provides a simplified implementation of the popular tic-tac-toe game. The game begins with the user making the first move by placing an "X" anywhere in a 3°×°3 grid. At this point the computer generates the graph of all possible moves and selects the one giving it the best chance of winning. Since some of the reviewed moves are no longer necessary, the other graph branches will be removed to release the allocated memory. While this is not absolutely necessary for game play, it serves as an example of recursive function invocation coupled with dynamic memory operations and pointer manipulation.

The fundamental data structure used in the program is *move_t* (lines 10—15), storing the details of a single move (placement of "X" or "O") in the game. It is a C structure containing the *next* and *level* pointers to other like structures, and permits building graphs extending in two dimensions. The *next* pointer points to a structure containing the immediate follow-on move from the current state. Since there may be more than one move possible, they are stored in the list linked by *level* pointers. The *wins* and *losses* fields contain the sum of all wins and losses computed for the entire subtree below the current move. A leaf node in the graph is one that ends with one of the parties winning (and hence one of the *wins* and *losses* fields is 0 and the other is 1) or a draw when the board is completely filled with no winner identified. For the latter both *wins* and *losses* are 0. The structure also contains the column and row coordinates of the current move (*x* and *y*), as well as information about *who* was moving (a character field storing either "X" or "O").

The move space is generated by the *build_tree* function starting at line 54. This function allocates a new *move_t* structure for every empty field found on the board, adding it to the *level* list. The list is pointed to by the *next* pointer stored in the upper-level node. If any of the board layouts in the list is identified as a winning move, the *wins* and *losses* fields are filled out appropriately. Otherwise, *build_tree* is invoked recursively, with the current move set as the ancestor node. In either case the values of wins and losses for the current move are added to the corresponding fields of the upper node, ensuring the propagation of these values up to the starting node.

Selecting the next computer move (*make_next_move*, line 120) involves traversing the level list immediately below the node representing the most recently made move. The strategy is rather

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 616

REFERENCE

[1] https://it.wikipedia.org/wiki/Trasformata_di_Fourier_veloce. Wikipedia, FFT, [Online].

ESSENTIAL LINUX

B.1 LOGGING IN

Most computers, including large installations, have various means to protect the stored information from unauthorized access. This is particularly important in systems that are shared by many users. The first line of defense consists of validating the access rights of a particular user, completely disabling access to storage contents and preventing the usage of system utilities when a user's identity cannot be properly confirmed. This is accomplished through the so-called login screen depicted in Fig. B.1. While the actual appearance may differ from system to system, the screen contains two fields that must be filled out. The first is the user identifier as assigned by the system administrator, which is a combination of letters, digits, and underscore ("_") and may be in some cases derived from the actual user's name. It has to be a unique, contiguous string of characters. The second entry accepts the user's password, or a secret combination of arbitrary printable characters, preferably including upper- and lower-case letters, digits, and punctuation marks. Note that the typed characters are replaced on screen by asterisks or dots to avoid showing the actual password text. For improved security, use of plain English words as listed in a dictionary should be avoided as much as possible. Many systems have rules governing password selection with which all users must comply, including the minimum password length. Of course, the user is obliged to remember his or her ID and password, and avoid disclosing the latter to anyone (including system administrators, since they have other means at their disposal to manage the user's account).

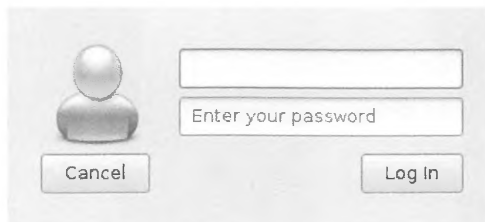


FIGURE B.1

Login dialog window used by a variant of Debian Linux distribution.

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.15002-6
Copyright © 2018 Elsevier Inc. All rights reserved.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 618

After entering the correct user name and password and clicking on the "log in" button (or its equivalent), the user is presented with a graphical desktop. To take advantage of material described in this book, a terminal application must be invoked. Linux distributions typically offer several such applications that differ in their look and feel, configuration options, and capabilities. The most common include *xterm* (a basic terminal emulator for the X Window system), *urxvt* (a Unicode-capable version of the older *rxvt* terminal), *gnome-terminal* (available in the Gnome desktop environment), and *konsole* (a terminal emulator bundled with the KDE desktop environment). They may be found in the "system" entry of the "application" menu on most desktops. Clicking on the relevant entry opens a terminal window with a prompt (usually a ">" or "$" character, on occasion following some additional information such as current time or host name), at which point one enters commands to be executed. A snapshot of a graphical desktop with an opened terminal emulator is shown in Fig. B.2.

In some cases a graphical desktop may not be available when using a simple text-based console or when more advanced display modes have not been configured or were disabled on the machine. This does not make it unusable, but the operation may be limited to text mode on a single terminal (see Fig. B.3). The login proceeds as described previously, with the user entering the credentials at the appropriate prompts. The main difference is that during the password entry no characters are echoed to the screen.

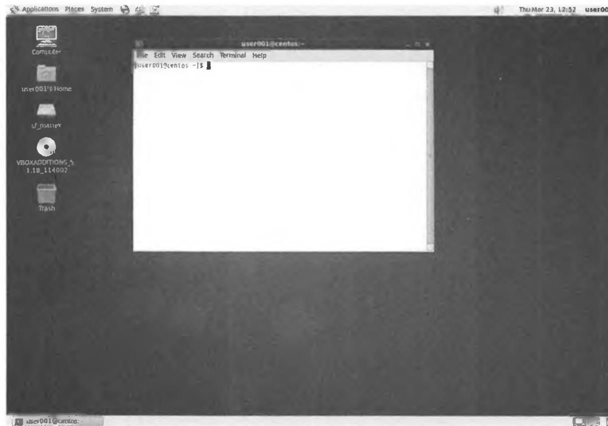


FIGURE B.2

Graphical desktop with a terminal emulator window (CentOS distribution).

(A)

(B)

FIGURE B.3

Terminal login: (A) authentication prompt; (B) shell prompt after successful user authentication.

B.2 REMOTE ACCESS

The login procedure delineated above may be used to gain access to a local Unix (or Linux in particular) machine. However, permitting only local login to a supercomputer would be overly restrictive. The preferred approach is to enable remote login over the network to accommodate even the broadest base of users without forcing them to be in physical proximity to the machine.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 620

The commonly used program for that purpose is Secure Shell, or *ssh*. It transmits all information, including login credentials, in encrypted format, thus preventing the potential eavesdroppers on the network from intercepting any useful data. A sample first-time *ssh* login sequence which may be executed on any terminal mentioned in Section B.1 is shown below:

```
> ssh user001@bigiron.some.university.edu
The authenticity of host 'bigiron.some.university.edu (10.0.0.1)' can't be
 established.
ECDSA key fingerprint is SHA256:pyYRR3L9EZXA1alJ/EyteIkfL2RJhwiAS2j174UbM1c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added "bigiron.some.university.edu" (ECDSA) to the list of
 known hosts.
user001@bigiron.some.university.edu's password:
[user001@bigiron ~]$
```

In this example the user name is "user001" and the login host is "bigiron.some.university.edu". Since this is the first time this user attempts to connect to "bigiron.some.university.edu", *ssh* warns that it does not have any previously collected information about the target machine. Unless there are good reasons to assume that host spoofing is indeed taking place, the user should answer "y" to the question about whether to continue connecting to the server. This produces the password prompt. Entering the password (which will not be shown on the screen) completes the login process by presenting the user with a shell prompt on the *remote* machine. The subsequent login sequence is much shorter, since the client machine is already in possession of the encrypted keys identifying the remote host. The warnings may reappear when the target machine changes its keys, which should happen relatively infrequently. As demonstrated in the example below, *ssh* also reports the time and origin of the last session to alert the user if a third party managed to gain illicit access to his or her account.

The connection to the remote host established in this way permits the use of arbitrary command-line applications that do not require a graphical environment to run. Occasionally, however, it is desirable to start a graphical user interface (GUI) application on the remote host to take advantage of improved user interface or to access graphical tools that are licensed only for local use. Adding a `-Y` option to the *ssh* command activates the X Windows protocol forwarding over the network connection:

```
> ssh -Y user001@bigiron.some.university.edu
Last login: Fri Mar 24 10:47:22 2017 from mybox.some.university.edu
[user001@bigiron ~]$ vtk
```

This results in the *vtk* window appearing on the client machine's display. Beware that slow network links may result in substantial delays between the command being issued and the resultant window being displayed (or even redrawn) on the client screen. Of course, the forwarding works only if the client machine supports and is actively running a graphical desktop environment.

B.3 NAVIGATING THE FILE SYSTEM

Persistent information in computer systems is stored in *files*, named entities organized in a hierarchy referred to as a *file system*. Unlike memory contents, once committed to physical storage the files survive machine reboots and shutdowns. All information that needs to be preserved, such as important datasets or programs to be executed, has to be stored in files. Files are explicitly created, read, and written by programs.

The file system provides its own naming scheme to facilitate access to specific files. Individual files have their own names, such as "*main.c*" or "*task_plan.txt*". While not strictly required, these names have typically two components separated by a period ("."): *base name* and *extension*. The base name may be anything the user has selected to hint at the file's purpose or suggest its contents. The extension describes the file's type. Thus for the two examples given above, "*main.c*" most likely contains source code of a program in C language, while "*task_plan.txt*" is probably a plain-text file containing the description of some planning activity. File names tend to be relatively short, as most file systems impose a fixed limit on their length. Since Unix file names may contain multiple periods, only the group of characters following the last of them is considered an extension.

Using plain file names as the only access method would quickly become unmanageable in systems that must handle thousands of them. It also does nothing to prevent collisions on the same file name. For example, to store two code versions of the same program source, one would have to be renamed; that change would in turn have to be propagated through scripts that build the final executable code, preventing reuse and creating unnecessarily replicas. To organize the file system contents better, the notion of *directory* was introduced. Directories are named groups of files and other directories. They may have (almost) any name, but unlike files they do not use extensions, although periods are permitted in their names. Directories may be arbitrarily nested. The top-level directory indicates the *root* (topmost entry) of the file system and is referred to as "*/*" (single forward slash). Forward slashes are used as separators of hierarchy levels leading to the specific file or directory. Thus "*/home/user001/src/heat.c*" uniquely identifies file named "*heat.c*", presumably a C source code, that is contained by the "*src*" directory stored in the "*user001*" directory, and that in turn resides in the "*home*" directory directly below the root of the file system. Note that this scheme allows for two or more different files called "*heat.c*" to exist within the file system as long as they are stored in different directories. The sequence of hierarchy elements necessary to reach the specific file starting from the root is called a *path*.

Using full paths to identify files being accessed may become tedious and verbose in the long run. Since commands are executed by the shell (see Section B.5), a more convenient solution is possible thanks to the notion of a *working directory*. The shell retains the path name of the working directory between command invocations, and updates it only when dedicated commands changing its value are called by the user. The working directory acts as a prefix that is prepended to the so-called *relative* paths, which are distinguished from full or *absolute* paths by the fact that they do not begin with a slash. Thus if the current working directory is set to user001's home, or "*/home/user001*", the C source file from the example may be referred to by typing only "*src/heat.c*". Other conveniences include syntax shortcuts that simplify the formation of various paths.

- The user's home directory is abbreviated to "~" (tilde character). The home directory is the current working directory set immediately following user login. Thus the file above could also be reached by using "*~/src/heat.c*".

- The current directory is represented by "." (single period). Thus the paths "*/home/user001/src*" and "*/home/./user001/src/.*" are equivalent. While it may look like this shortcut is not terribly useful, its benefits become more obvious when discussing commands that take directories as arguments.
- The parent directory may be expressed as ".." (double period). Hence "*/home*" and "~/.." refer to the same directory.

File systems typically conform to default layout rules to make computer users more effective and help them to find various utilities, documentation, and appropriate storage for data. See , Chapter 3 for a more detailed description of the standard directory structure utilized by Linux distributions.

A number of Linux system utilities have been developed, with a goal of simplifying the management of the file and directory hierarchy and accessing file contents. They are briefly described below, with simple usage examples. The shell prompts were shortened to a single ">" character for brevity. Due to context sensitivity, the examples were created with the assumption that they are executed in order on the same host.

- ***ls*** (list directory contents)

  Without arguments, this lists the contents of the current working directory. The arguments may include an arbitrary number of file and directory paths. Additional options may provide more information about the stored entries, such as access permissions, ownership, modification date, size, etc. The commonly used ones include "-l" to select the "long" format and "-a" to enable display of hidden entries (i.e., all files and directories whose names start with a period).

  Note: many distributions by default alias the *ls* command with frequently used options to "*ll*".

```
> ls -l /home/user001/src
total 8
-rw-r--r-- 1 user001 user001 361 Mar 24 17:55 Makefile
-rw-r--r-- 1 user001 user001 491 Mar 24 17:55 heat.c
```

- ***cd*** (change working directory)

  This changes the working directory to the specified path or, when invoked without arguments, to the user's home directory. The example below changes the working directory to the user's home, descends into the "*src*" subdirectory, and lists all files there:

```
> cd
> cd src
> ls -la
total 16
drwxr-xr-x 2 user001 user001 4096 Mar 24 20:03 .
drwxr-xr-x 3 user001 user001 4096 Mar 24 20:07 ..
-rw-r--r-- 1 user001 user001  361 Mar 24 17:55 Makefile
-rw-r--r-- 1 user001 user001  491 Mar 24 17:55 heat.c
```

- ***pwd*** (print working directory)

  This prints the current working directory.

```
> pwd
/home/user001/src
```

- ***mkdir*** (make directory)

  This creates new directories whose paths are specified as arguments to the command. Nominally, the new directory must be an immediate child of an existing path. To permit creation of arbitrarily nested paths, a "`-p`" option (for "parents") should be used. The example below has to use the parents option, since the "*src2*" directory currently does not exist:

```
> mkdir -p ~/src2/tmp
> ls -l ../src2
total 4
drwxrwxr-x 2 user001 user001 4096 Mar 24 20:44 tmp
```

- ***cp*** (copy files and directories)

  This takes at least two path arguments: the last argument is the destination for the copy operation, while all preceding arguments are considered to be the source arguments. Source arguments must exist. Multiple sources are permitted only if the destination is an existing directory. To copy source directories properly, a recursive option "`-r`" should be specified.

```
> cp heat.c ~/src2/heat2.c
> cp -r ~/src2 .
> ls -l src2
total 8
-rw-r--r-- 1 user001 user001  491 Mar 24 21:24 heat2.c
drwxrwxr-x 2 user001 user001 4096 Mar 24 21:24 tmp
```

  The example above copies the content of the *heat.c* file to the *src2* directory created before and stores it in a file named *heat2.c*. The second call copies the whole directory "/home/user001/src2" to the current working directory (note the use of "."). Since the whole subdirectory tree is replicated, a recursive option is used.

- ***mv*** (move files and directories)

  The move command is used to relocate files and directories within the file system. Its syntax resembles that of *cp*, but the recursive option is no longer necessary, since changing the location of

a directory implies changing it for all its children. The *mv* command may also be used to rename either files or directories.

```
> mv src2/heat2.c src2/tmp
> mv src2/tmp ./other
> ls -l other src2
other:
total 4
-rw-r--r-- 1 user001 user001 491 Mar 24 21:24 heat2.c
src2:
total 0
```

The first command moves file *heat2.c* from the *src2* directory to its child subdirectory *tmp*. The second moves that subdirectory along with its contents to the current working directory and renames it to "*other*". The directory listing confirms that the operations have been carried out correctly: the *src2* directory is now empty and "*other*" directory is now a direct child of the current working directory and contains file *heat2.c* that was originally stored in *src2*.

- ***rm*** (remove files of directories)

  The *rm* command irreversibly deletes files or directories. For the latter, a recursive option ("`-r`") must be added. The example shows how to remove the now empty *src2* subdirectory:

```
> rm -r src2
> ls -l
total 12
-rw-r--r-- 1 user001 user001  361 Mar 24 17:55 Makefile
-rw-r--r-- 1 user001 user001  491 Mar 24 21:38 heat.c
drwxrwxr-x 2 user001 user001 4096 Mar 24 21:51 other
```

- ***find*** (look for specific files)

  The *find* command searches for files of predefined characteristics. While its option list is quite extensive, it is frequently used to find files or directories with specific names. This is controlled by the predicate "`-name` *filename*" for exact matches and "`-iname` *filename*" for case-insensitive matching. The search may be further restricted to report only directories by specifying "`-type d`" and regular files by using "`-type f`". The only argument of *find*, immediately following the command, is a path name identifying the top directory on whose contents the lookup will be performed recursively. The example below attempts to find all file system entries named *heat2.c* that exist at any hierarchy level under the user's home directory, and later all files starting with "heat" to demonstrate wildcard use (explained in detail

in Section B.5). Note that the latter requires protecting the name argument from shell expansion by enclosing it in single quotes:

```
> find ~ -name heat2.c
/home/user001/src/other/heat2.c
/home/user001/src2/heat2.c
> find ~ -name 'heat*'
/home/user001/src/heat.c
/home/user001/src/other/heat2.c
/home/user001/src2/heat2.c
```

B.4 EDITING THE FILES

Having learned the basics of file system access, the next step is to create files with the desired contents. This capability is provided by text editors. Linux distributions offer many options of different complexity, resource footprint, supported environments, and integration features targeting code development. One of the main distinguishing factors for editor selection is GUI availability: some editors may only work invoked inside a text terminal, some support graphical desktops, and a subset provides both. For editors that are incapable of accessing remote files, terminal-based operation consumes much less network bandwidth, resulting in smoother editing when invoked on a remote machine (such as a supercomputer's login node). A short description of commonly used text editors is given below.

B.4.1 VI

Vi is a nonGUI editor with a broad user base and a long history in Unix environments originating in the 1970s. Its code has been updated many times and has inspired a number of clones. Perhaps the most characteristic feature of vi is modality. Text input using keystrokes (insert mode) and execution of editor commands (normal or command mode) are performed in dedicated modes of operation that the user explicitly switches between. Most current Linux versions bundle vim ("Vi IMproved"), which offers new features compared to the original, such as syntax highlighting (coloring of various syntactic constructs in programming languages to make the code makeup more apparent), mouse support, completion, file comparison and merging, regular expressions, scripting, spell checking, tab support, and many others. Vim is also available in a GUI variant, complete with menus and toolbars, as gVim.

B.4.2 EMACS

Another editor with an established presence in the Unix world is Emacs. Its name was derived as a contraction of "editor macros". Of a number of clones spawned over the years, the most popular remains GNU Emacs, a free software implementation based on a small core written in C with most

functionality provided by the Elisp (a dialect of LISP) extension language. GNU Emacs layout consists of a main text window and a much smaller minibuffer that displays status information and acts a command interface. This layout works in both text-only mode and with a GUI, although the latter also provides a set of menus for common operations. The editor is highly extensible and configurable, implementing over 2000 commands. Another important feature is support of major and minor modes, in which a specific major mode is activated per file buffer and usually triggered by the file type (such as C code or HTML source), while further customizations, including on-the-fly spell checking, automatic line breaking, or highlighting of specific portions of text, may be enabled or disabled at any time. Any number of minor modes may be active at any time.

B.4.3 NANO

The GNU nano editor embraces interface simplicity as its main design goal, making it an obvious choice for beginners. The commonly used command key combinations are displayed on the same screen as edited text, thus not requiring their memorization for effective editing. Despite its small size, nano features colored text, multiple edit buffers, search-and-replace operations based on regular expressions, recent operation undo and redo, and modification of key bindings. Nano operates in text-only mode.

B.4.4 GEDIT

Gedit is a GUI-centered editor developed for the Gnome desktop environment, deployed by default on many Linux distributions including Ubuntu, Fedora, Debian, CentOS, and others. The editor's long list of features supports syntax highlighting, multilanguage spell checking, tabbed mode, session preservation, line numbering, parenthesis matching, automatic indentation, autosave, font configuration, etc. Gedit is also capable of editing files on remote hosts. Its core functionality may be further extended through plugins.

B.4.5 KATE

The other popular desktop environment, KDE, provides a default GUI-based editor called Kate. The basic feature set is similar to that of Gedit. Kate's indentation and tool functionality accessed through the command line may be additionally customized through javascript code. Kate is the source code editor used by the Kdevelop integrated development environment in KDE.

File editing using any of these editors is quite straightforward. The editor programs are started by typing their name (in lower case) on the command line, optionally followed by the path name of the file one wants to modify or create. The editor marks the place being edited with a *cursor*, or a single character-sized block or bar that may be blinking for faster visual location on the screen. The arrow keys on the keyboard move the cursor in any of the four principal directions (up, down, left, and right). Typing printable characters enters them at the cursor's current position and displaces the existing text to the right. Advancing the text by larger strides is possible using the page-up and page-down keys in most editors. With the exception of vi, the editors allow free mixing of text editing and command execution. In vi the insert mode (for typing the text directly) is activated by typing letter "i" in command mode, while switching back to command mode

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 627

Table B.1 Key Bindings for Frequently Used Commands in Popular Text Editors

Function	Vi	Emacs	Nano	Gedit	Kate
Display help	:h	Ctrl-h	Ctrl-g	F1	F1
Undo	:u	Ctrl-x u	Alt-u	Ctrl-z	Ctrl-z
Open a file	:r *filename*	Ctrl-x Ctrl-f	Ctrl-r	Ctrl-o	Ctrl-o
Save file	:w	Ctrl-x Ctrl-s	Ctrl-o	Ctrl-s	Ctrl-s
Save as another file	:w *filename*	Ctrl-x Ctrl-w	Ctrl-o	Ctrl-Shift-s	Ctrl-Shift-s
Find a string	/	Ctrl-s	Ctrl-w	Ctrl-f	Ctrl-f
Search and replace	:s/*pattern*/*replacement*/	Esc %	Alt-r	Ctrl-h	Ctrl-r
Cut text	dd	Ctrl-k	Ctrl-k	Ctrl-x	Ctrl-x
Paste text	P	Ctrl-y	Ctrl-u	Ctrl-v	Ctrl-v
Exit	:q	Ctrl-x Ctrl-z	Ctrl-X	Ctrl-q	Ctrl-q

All vi commands must be entered in the "normal" (command) mode. For other editors, Ctrl, Alt, Shift, and F1 denote specific keys on the keyboard. A dash following one of these keys and a letter signifies concurrent activation of several keys. For example, the Ctrl-h sequence is performed by first pushing the control key and then depressing the "h" key while the control key is held down.

is performed by pressing the escape key. Table B.1 summarizes the keyboard shortcuts for common editing operations.

B.5 ESSENTIAL BASH

Command invocation, job management, and many aspects of file handling may be vastly simplified by using various features provided by the shell. The shell is used to issue commands, display their output, and manage concurrent tasks. It also acts as an interpreter for a language that may express sequences of operations and implement elements of flow control that permit building custom execution scripts. Since *bash* is set up as the default login shell when creating new user accounts and is configured by default by many (if not all) Linux installations, this section focuses exclusively on its syntax and features.

B.5.1 PATH EXPANSION

The first important feature that permits easier manipulation of file groups is called path expansion. The characters "*" (asterisk), "?" (question mark), "[', ']" (square brackets), and "{', '}" (curly braces) have special meanings when used inside directory and file paths. The first matches any string of characters, including an empty string. Assuming the current working directory contains files as listed by the command below:

```
> ls
Makefile  example.txt  f1.txt  f2.txt  f22.txt  heat.c
```

matching all files with the extension ".txt" may be accomplished as:

```
> ls *.txt
example.txt  f1.txt  f2.txt  f22.txt
```

The question mark matches precisely one character. Thus to select only files from the above set with a single character between the "f" character and the extension ".txt", one could type:

```
> ls f?.txt
f1.txt  f2.txt
```

Square brackets match one of the specified characters. For example, to find files with the second letter in their names of either "a" or "e", the following pattern may be used:

```
> ls ?[ae]*
Makefile heat.c
```

Braces are used to list arbitrary substrings or patterns to be matched. Multiple substrings must be comma separated. Thus the following selects all files for which the ".txt" extension is immediately preceded either by a digit "2" or a string "ple":

```
> ls *{2,ple}.txt
example.txt f2.txt f22.txt
```

To verify that patterns work when specified within braces, try to match all text files whose base names are exactly two characters long or end in "ample":

```
> ls {??,*ample}.txt
example.txt f1.txt f2.txt
```

The path substitution forms discussed above may be applied to any portion of a path name, including directory components. However, the matching is always limited to the single level of the hierarchy. Thus "/*" will not select all entries present in the file system, but only files and directories contained by the root directory.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 629

B.5.2 SPECIAL CHARACTER HANDLING

On occasion it may be necessary to refer to a path name that contains one or more special characters. In such situations these characters will have to be escaped using a backslash ("\") or placed between single quotes. To refer to a file named "ready?", the actual string argument appearing on the command line would have to be typed as "ready\?" (without the double quotes which serve here only as name delimiters) or "'ready?'". As the shell breaks down the command-line contents at blank spaces (which actually may be regular spaces or tabs) to identify command options and arguments, the same method may be used to refer to files whose names contain spaces. Shell language syntax assigns special meaning to several other characters, making escaping them necessary if used within path names. They include the pipe symbol ("|"), ampersand ("&"), semicolon (";"), parentheses ("(' and ')"), angle brackets ("<" and ">"), and the end-of-line character. To minimize the occurrence of related problems, avoiding use of these characters in file names is a good general rule, especially for users who are just beginning to learn the shell concepts.

B.5.3 INPUT/OUTPUT REDIRECTION AND PIPELINES

Some commands executed by the shell generate output and some expect input data. The shell provides dedicated operators to manage the standard input, standard output, and standard error streams mentioned in Appendix A. In Unix systems these streams are by convention associated with file descriptors numbered 0—2, respectively. So-called "redirection" may be used to channel the input to the application from a specific file (instead of requiring the user to type in the input data every time that application is run) or permit the capture of application output in a file (instead of making just an ephemeral appearance on the screen). The operators governing the I/O redirection include the following.

- ">" redirects the application's standard output to the specified file. To illustrate this, a *cat* utility (described in Section B.7.1) is used to display the contents of the file:

```
> ls *.c > c_files
> cat c_files
heat.c
```

- "&>" redirects both standard output and standard error to the specified file. The example below tricks *ls* into generating error output by specifying an argument that refers to a nonexistent file:

```
> ls *.h > h_files
ls: cannot access *.h: No such file or directory
> cat h_files
> ls *.h &> h_files
> cat h_files
ls: cannot access *.h: No such file or directory
```

Since “>” only redirects the standard output, the error was not captured as the content of “*h_files*” but displayed in the terminal instead. The capture file is still created, but nothing is stored in it. The second invocation applies the “&>” operator that redirects both types of output. This is particularly useful when saving the output of installation or compilation scripts; using only the first kind of redirection might omit the actual error information, making the subsequent troubleshooting more difficult.

- “>>” redirects the standard output while appending it to the specified file. This variant may be used to merge the output of several commands into a single file, since the application of “>” to the same file would simply overwrite its contents. To illustrate this in action, a shell built-in *echo* command that outputs (echoes) strings to the standard output will be used. Since path expansion is applied to all unescaped arguments in commands executed by the shell, there is no need to invoke the *ls* command explicitly:

```
> echo "These are my C files:" *.c >> my_files
> echo "These are my text files:" *.txt >> my_files
> cat my_files
These are my C files: heat.c
These are my text files: example.txt f1.txt f2.txt f22.txt
```

- “&>>” redirects both standard output and error streams to be appended to the given file.
- “<” redirects the application’s standard input to be read from the specified file. The somewhat contrived example below (since the *cat* utility can accept a file argument directly) demonstrates its use:

```
> cat < my_files
These are my C files: heat.c
These are my text files: example.txt f1.txt f2.txt f22.txt
```

As many applications both accept input and generate output, it stands to reason there is a way to daisy-chain them to implement more complex processing flows. This concept is called pipelining, and is realized using the pipe operator “|”. It enables forwarding the standard output created by command k to be standard input of command $k+1$ in the pipeline, as shown below:

```
command_1 | command_2 | ... | command_n
```

Of course, the output(s) of *command_n* may be captured in a file by applying the redirection mechanism described above. A variant of the pipe operator, “|&”, supports redirection of merged standard output and error streams to the standard input of the next pipeline stage.

B.5.4 VARIABLES

Bash supports variables that may be used to store arbitrary strings produced by commands and applications or to retain control state in scripts. The fundamental variable assignment statement is

name=value

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 631

where *name* is a variable identifier consisting of an arbitrary combination of letters, digits, and underscores as long as the first character is not a digit. *Value* may be a string or an array. It may also be omitted, in which case an empty variable is created. Once assigned, the variable value can be retrieved by placing a dollar sign ("$") before its identifier:

```
> x=99
> echo $x
99
```

Array variables may be created by the explicit assignment of an element at a specific index, such as

name[*index*]=*value*

where *index* has to evaluate a number. Another way to create arrays is by assignment of list of values:

name=(*value_1 value_2 … value_n*)

To dereference a specific element of an array, ${*name*[*index*]} format should be used. Special subscripts of "@" or "*" retrieve all values of the array, but with one difference: the first produces a list of values very much mirroring the way they were originally assigned to the array, while the latter yields a single string containing concatenated values. The first element of an array is located at index 0, hence:

```
> numbers=(one 2 three 4 five)
> echo $numbers
one
> echo ${numbers[@]}
one 2 three 4 five
> echo ${numbers[2]}
three
```

The content of an array may be expanded using the "+=" operator. For instance:

```
> fruits=(apple peach)
> fruits+=(banana)
> echo ${fruits[@]}
apple peach banana
```

Defined variables are normally accessible within the scope of the current shell. Since shell scripts are executed in a subshell, they typically do not have access to the parent shell variables set up in the way described above. To enable such access, each variable must be explicitly exported.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 632

This is accomplished by preceding the variable assignment (or just the variable name if already defined) by the keyword "export":

```
> cat showx
#!/bin/bash
echo $x
> x=99
> ./showx

> export x
> ./showx
99
```

In the example above, the first line of the script "*showx*" is a hint to the execution environment that the remainder of the file should be interpreted by a program residing at the specified path ("/bin/bash" in this case). The variable "x" is unknown to the script until it is exported. The same could have been accomplished at the variable definition time using "`export x=99`".

Variables may be deleted using a statement in the form:

unset *name*

```
> echo $x
99
> unset x
> echo $x
```

Bash provides a number of predefined variables that may provide useful information to scripts. The necessarily limited list below describes those most often used.

- **BASH** provides the full path name leading to the shell program currently executing.
- **BASHOPTS** lists enabled shell options in a colon-separated format.
- **BASH_VERSION** gives the version number of the currently executing shell.
- **HOSTNAME** contains the name of execution host.
- **MACHTYPE** describes the system (machine) type the shell is running on.
- **OSTYPE** identifies kind of operating system executing on the host.
- **PATH** contains a colon-separated list of directory locations (search paths) that the shell scans for commands. For any command invoked by name only (i.e., without specifying the path to its executable), Bash will try to determine its location by checking each specified search path in the order listed.
- **PWD** is the path name of the current working directory.

- **OLDPWD** is the path name value identifying the previous working directory.
- **GLOBIGNORE** contains a colon-separated list of patterns to be ignored when performing path name expansion.
- **HOME** stores the path name of the user's home directory.
- **GROUPS** is an array of identifiers of groups of which the user is a member.
- **PIPESTATUS** is an array storing exit status values of all processes comprising the most recently executed pipeline statement.
- **RANDOM** is a variable generating a random integer value between 0 and 32767 whenever read.
- **SECONDS** stores the number of seconds elapsed since the shell was started.

Bash supports many syntactic enhancements that provide additional information about existing variables or transform them into other forms of data. The commonly encountered constructs include the following.

- "**${#*name*}**" returns the length of the variable (the number of characters used by its string representation).
- "**${#*name*[@]}**" provides the count of elements stored in the array.
- "**${*name*:*offset*}**" or "**${*name*:*offset*:*length*}**" performs substring expansion, i.e., it extracts the section of string of *length* characters starting at an *offset*. If *length* is not specified, the substring starts at an *offset* and continues until the last character of *name*.
- "**#{*name*/*pattern*/*string*}**" substitutes the first longest occurrence of *pattern* with *string*. If *pattern* begins with "/", every occurrence of *pattern* is replaced. The *string* may be empty, in which case the second "/" may be omitted.
- "**${*name*#*pattern*}**" or "**${*name*##*pattern*}**" removes the matching prefix. The first deletes the shortest matching prefix, while the second form removes the longest. The pattern is transformed using path name expansion rules.
- "**${*name*%*pattern*}**" or "**${*name*%%*pattern*}**" is analogous to the previous construct except it removes the suffix portion of the string.

Examples:

```
> s=/home/user001/error.c
> echo ${#s}
21
> echo ${s/er/ing}
/home/using001/error.c
> echo ${s//er/ing}
/home/using001/ingror.c
> echo ${s##*/}
error.c
> echo ${s%/*}
/home/user001
```

B.5.5 ARITHMETIC ON VARIABLES

Variables representing numbers may be used in simple arithmetic expressions. The construct to accomplish this is "$((*expression*))" and may be nested.

```
> x=99
> echo $(((x+1)*10))
1000
```

The supported operators include "+" (addition), "-" (subtraction), "*" (multiplication), "/" (division), "%" (remainder), "**" (exponentiation), "~" (bitwise negation), "&" (bitwise and), "|" (bitwise or), "^" (bitwise exclusive or), "<<" (left bitwise shift), ">>" (right bitwise shift), "==" (compare for equality), "!=" (compare for inequality), "<" (less than), "<=" (less or equal), ">" (greater than), ">=" (greater or equal), "&&" (logical and), "||" (logical or), "*expr1?expr2:expr3*" (conditional operator), "*name++*" (postincrement), "*++name*" (preincrement), "*name–*" (postdecrement), and "*–name*" (predecrement). The last four operators change the value of variable *name*. While the post- variants return the variable value before the operation is performed, the pre-variants return the value of variable after the update. For example:

```
> echo $x
99
> echo $((x++))
99
> echo $x
100
```

B.5.6 COMMAND SUBSTITUTION

A particularly useful feature of the shell is the ability to capture directly the output of a command in a variable. There are two forms of syntax to do this: by encasing the command in a pair of backquotes ("`"), or by invoking it as "$(*command*)". The command may be compound, including a pipeline. Bash provides a faster option to read file contents into a variable with "$(< *file*)" rather than "$(cat *file*)". The following example stores paths of all files matched by the *find* command into the variable "text_files":

```
> text_files=`find . -name "*.txt"`
> echo $text_files
./f22.txt ./f2.txt ./example.txt ./f1.txt
```

B.5.7 CONTROL FLOW

Creation of sophisticated shell scripts takes advantage of more complex constructs that permit definition of loops and conditional execution. To introduce them, the concept of exit status needs to be explained. Every command and application run by a shell returns a numeric status value when

it finishes the execution; this value is not displayed on the screen but kept internally by the shell. For C programs, this is the value of the expression following the "return" keyword in the main function or argument to the "exit" library function. For shell scripts, the status is that of the last command executed by the script, or zero if no commands were executed. By convention in Unix systems, zero exit status indicates success, while any nonzero value is a failure.

A brief overview of the commonly used constructs is presented below.

- "*command_1* ; *command_2* ; … ; *command_n*" executes each of the specified commands in order, waiting for command x to finish before starting the command x+1. The exit status is that of the last command.

```
> ls M*; ls *.h; ls *.c
Makefile
ls: cannot access *.h: No such file or directory
heat.c
```

- "*command_1* **&&** *command_2* **&&** … **&&** *command_n*" executes commands in sequence, stopping after the first failing command. The example below attempts to list different kinds of files and displays "Success!" if all of them exist:

```
> ls *.c && ls M* && echo "Success!"
heat.c
Makefile
Success!
```

- "*command_1* || *command_2* || … || *command_n*" attempts to execute command *k+1* only if all *k* preceding commands failed. None of the commands following the successful one is executed. For example:

```
> ls *.h || echo "Could not find any files!"
ls: cannot access *.h: No such file or directory
Could not find any files!
```

- "**for** *name* **in** *word1 word2* … **; do** *list* **; done**" implements a loop that iterates over values represented by *word1*, *word2*, etc. while storing them in a variable *name*. That variable may be referenced by any of the commands inside the loop body represented in the syntax above by *list*. For example:

```
> for f in 'ls *.txt'; do echo "Text file:" $f; done
Text file: example.txt
Text file: f1.txt
Text file: f2.txt
Text file: f22.txt
```

- "**for** ((*expr1* ; *expr2* ; *expr3*)) ; *list* ; **done**" implements an arithmetic loop which resembles the "for-loop" syntax in C language discussed in Appendix A. For instance:

```
> for ((x=2; x<5; x++)); do echo "square of $x is $((x*x))"; done
square of 2 is 4
square of 3 is 9
square of 4 is 16
```

- "**while** *list1* ; **do** *list2* ; **done**" behaves similarly to the while-loop in C language. As long as the status of the last command in *list1* is zero, commands in *list2* are executed. The exit status is that of the last executed command in *list2*, or zero if none was run. The following sequence of commands appends paths in reverse order from the "files" array to the "names" array until the latter includes four elements or there is nothing left to copy:

```
> files=(*.txt *.c Makefile)
> echo ${#files[@]}
example.txt f1.txt f2.txt f22.txt heat.c Makefile
> names=() > i=${#files[@]} > while ((${#names[@]}<4 && $i>0)); do names+=(${files
 [$((--i))]}); done
> echo ${names[*]}
Makefile heat.c f22.txt f2.txt
```

- "**if** *list1* ; **then** *list2* ; [**else** *list3* ;] **fi**" executes statements in *list2* if the exit status of *list1* is zero. Otherwise, if the **else** branch is specified, statements of *list3* are executed. For example:

```
> for ((x=3; x<6; x++)); do echo -n "cube of $x is "; if ((x**3%2==0)); then echo even;
 else echo odd; fi; done
cube of 3 is odd
cube of 4 is even
cube of 5 is odd
```

Note that the "-n" option to *echo* suppresses the output of the end-of-line character.

B.6 COMPILATION

Compilation is a process of converting the program description stored in one or more source files to executable code. Creation of executable files typically proceeds in two stages: generation of so-called object files for each C language source file, and linking the resultant files into a single final executable binary. Many compilers support invocation formats that permit combination of these two phases into a single command for convenience.

Object files normally have an ".o" extension. They contain machine instructions to be later executed by the CPU, but which cannot run by themselves. The "gcc" C compiler commonly found in Linux distributions uses the -c option to create them. Let us assume we have three source files

(as shown in the example below) that together contain the full functionality of the program. The "main" function is defined within the "main.c" file; the other sources may not contain their own "main" functions, since it would make it ambiguous as to which one of them is the entry point to the program. Compilation of the "main.c" file to object code is invoked as follows:

```
> ls
main.c src1.c src2.c
> gcc -c main.c
> ls
main.c main.o src1.c src2.c
```

Note that the object file created in that way retains the base name of the input source file, only replacing the extension. If the source code does not contain any problematic constructs or undefined identifiers, the compiler typically will not produce any text output. The code generation is controlled by a plethora of options, of which the most common are listed below.

- **-O***number* performs code optimizations at a level determined by *number*. Generally, the higher the level and the more involved the optimizations, the better the resultant performance of the code, but also the longer compilation. In practice, "-O2" and "-O3" offer the best tradeoff between compilation time and code quality. The "-O0" turns off the optimizations—this is the default behavior when no optimization option is specified.
- **-g** embeds the debugging information, such as variable and function names, in the resultant object file. While gcc permits combining debugging and optimization options in the same command, one has to remember that higher optimization levels may severely modify flow control in the program, on occasion completely eliminating some variables or functions. A variant of this option, **-ggdb**, produces debug information specifically for use with the GNU debugger, potentially including gdb-specific extensions.
- **-o** *file* places the compiler output in an explicitly named *file*. When generating an object file, it should have an ".o" extension.
- **-I***directory* adds *directory* to the set of header (files with an ".h" extension) search paths. Multiple -I options are permitted in the same command. Header files installed under "/usr/include" (such as prototypes and macros used by the C library) are searched by default.

To make the object files into a self-contained program, a linker must combine them together, make sure there are no missing functions and variables, and add extra code that correctly sets up the execution environment. Conveniently, the gcc compiler may be also used to perform this operation. Remembering that there two more source files to compile, and the remaining sequence of commands is as follows:

```
> gcc -c src1.c
> gcc -c src2.c
> gcc main.o src1.o src2.o -o my_program
> ls
main.c main.o my_program src1.c src1.o src2.c src2.o
```

The newly created executable "my_program" may be now invoked at the shell prompt just like any other program. If the "-o" option is not used, a default executable name is assumed, typically "a.out" on Unix systems.

In simple cases similar to the example above, creation of the executables may be performed in a single command. The intermediate object files are not retained in this case, so from the user's perspective it appears as though the compiler produced the final binary directly from sources. The following example illustrates this, while at the same time performing code optimization:

```
> rm -f *.o my_program
> gcc -O2 main.o src1.o src2.o -o opt_program
> ls
main.c opt_program src1.c src2.c
```

The examples so far have not created or taken explicit advantage of external libraries. Actually, the latter is not quite correct: the linker silently links the object code with the system's C library, so if any of the listed sources invoked C library functions or used its internal variables, they would be automatically resolved. To learn how to create custom libraries, let us assume that "src1.c" and "src2.c" contain functionality that could be reused by several programs and is thoroughly debugged and fine-tuned for performance. It would thus make sense to avoid their recompilation every time a new version of the program needs to be built. This is accomplished by converting them into a library, with the familiar first step involving compilation to object files with the desired debugging and optimization flags:

```
> gcc -c -g -O2 src1.c
> gcc -c -g -O2 src2.c
> ar rcs libmy_library.a src1.o src2.o
```

The last command invokes the Unix archive tool "ar" that packages all specified object code files into a library file named "libmy_library.a". Customarily, code libraries have the ".a" extension and carry names starting with "lib". The remaining, not yet compiled, functionality of the program is now limited to the contents of the file "main.c". To create a correctly formed executable, just two more commands are needed:

```
> gcc -c -g -O2 main.c
> gcc main.o -o opt_deb_program -L. -lmy_library
> ls
libmy_library.a  main.o           opt_program  src1.o  src2.o
main.c           opt_deb_program  src1.c       src2.c
```

As a result, an "opt_deb_program" optimized executable with debug symbols has been created. Note that the linking command this time contained only one object file "main.o", since the other required program functions are already provided by the library. To tell the linker which libraries should be used when looking for missing symbols, **-l***name* option is used, where *name* is the library file name stripped of the "lib" prefix and extension. Since custom libraries may reside anywhere in the file system, the linker is informed about their location through the −L*directory* option. Of course, the linking command may specify multiple library search paths and multiple libraries.

B.7 OTHER COMMAND-LINE UTILITIES

B.7.1 TEXT TOOLS

less (file viewing utility)

The *less* program is a simple file visualization tool, also called a pager, that permits scrolling of file contents by an arbitrary number of lines (using arrow keys), pages (page-up and page-down keys), and jumping directly to a specific location (line number followed by a "G"). The supported navigation and text search operations are a subset of the vi editor commands.

cat (concatenate files and print them on standard output)

This command takes any number of file arguments and merges their contents in a specified order. The concatenated text is printed to the standard output. When used without arguments, it passes standard input to standard output.

```
> cat f1.txt
file 1
> cat f2.txt
file 2
> cat f*.txt
file 1
file 2
```

head (print the beginning part of files)

The *head* command outputs the first number of lines ("-n *number*" option) or characters ("-c *number*" option) of specified files to the standard output. If the *number* is preceded by a minus, the output includes all but the last *number* of lines or characters. Without options, it prints the first 10 lines of indicated files. If multiple files are given, the printout for each is preceded by a header indicating the file name. The example below shows that no extra end-of-line character is added at the end of output (hence the shell prompt is adjacent to the printed text), and that end-of-line characters are included in the count.

```
> cat example.txt
line 1
line 2
line 3
> head -c 10 example.txt
line 1
lin>
```

tail (print the last part of files)

Analogous to *head*, this outputs the last *number* of characters or lines (the same options are used) of files. The *number* may be prefixed with a "+" (plus sign) to force starting the output with the *number*th character or line of the file. The *tail* command is also often used to monitor growing files (with contents appended by other running applications). This behavior is activated by option "-f".

```
> tail -n +3 example.txt
line 3
```

cut (cut a section of each line)

The command selects a specific range of characters (option "-c *list*") or fields (option "-f *list*") from each line of the input files (or standard input if "-" is specified instead a file name) and prints it to the standard output. The fields are determined by splitting each line at every occurrence of a predefined delimiter character (controlled by the option "-d *character*"), or by default the tabulation mark. The list may be a single integer to identify a specific field or character, a range in the form *start-end* (inclusive), or with the first or last number of the range missing, indicating starting from the first or ending on the last field or character of the line, respectively. The example below illustrates how to set the space character as the field delimiter:

```
> cut -f 2- -d ' ' example.txt
1
2
3
```

grep (find lines matching a pattern)

The *grep* utility matches lines that contain a specific character pattern. Its arguments include text pattern to look for, and optionally names of the files to search (standard input is assumed if no files are given). With multiple files, for each line containing the pattern *grep* outputs the name of the relevant file followed by the contents of the line. Printing of line numbers may be requested with option "-n". The matching is normally case sensitive, but specifying "-i" suppresses this behavior. *Grep* behavior may be reversed to output all the lines which do not contain the specified pattern by adding the "-v"

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 641

option. Finally, recursive searches on directories may be triggered with option "-r". The latter permits specification of directory paths as command arguments.

```
> grep -n 'e 2' f*.txt example.txt
f2.txt:1:file 2
example.txt:2:line 2
```

B.7.2 PROCESS MANAGEMENT

ps (output current process status)

Applications and system utilities that are not built-in shell commands have to be started as processes. To view a snapshot of their status, a *ps* command is used. Without options, it reports only processes that belong to the current user:

```
> ps
 PID TTY          TIME CMD
18441 pts/25 00:00:00 bash
18444 pts/25 00:00:00 ps
```

The processes are characterized by their process identifier (PID), a numeric handle that uniquely identifies the running process. To display all processes running in the system along with full information about them, "*ps auxw*" may be invoked (note there is no minus preceding the options). An interesting variant presented below reorganizes the output to display *process tree*, in which one can determine which processes are children of others:

```
> ps ax -H
  PID TTY   STAT   TIME COMMAND
...
    1 ?     Ss     0:04  /sbin/init
  370 ?     S      0:00    upstart-udev-bridge --daemon
  374 ?     Ss     0:00    /lib/systemd/systemd-udevd --daemon
  514 ?     S      0:01    upstart-socket-bridge --daemon
...
```

kill (deliver a signal to a process)

As its rather gruesome name suggests, the *kill* command may be used to terminate processes via Unix's signal mechanism. Not all signals result in a process termination; some may interrupt its execution, pause it, etc. Their full listing may be obtained with the "*kill −l*" command.

Without any options, a TERM (terminate) signal delivered to a process is in many cases sufficient to cause its more or less graceful termination. Some stubborn processes may ignore it, in which case a

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 642

KILL signal must be sent. The arguments of the *kill* command are PIDs of target processes. The example shows how to kill the user's *bash* process (which is usually a bad idea and is mentioned here only for illustrative purposes) with a PID obtained from the *ps* listing above:

```
> kill —KILL 18441
```

B.7.3 DATA COMPRESSION AND ARCHIVING

gzip (compress or expand a file)

The *gzip* utility is one of the most common compression programs, characterized by achieving substantial data compaction ratios (especially for text files) and fast operation. It takes as its argument path the name of the file or multiple files to be compressed:

```
> ls —l Makefile
-rw-r--r-- 1 user001 user001 361 Mar 24 17:55 Makefile
> gzip Makefile
> ls —l Makefile*
-rw-r--r-- 1 user001 user001 233 Mar 24 17:55 Makefile.gz
```

Gzip removes the original file if the compression is successful and adds the "*.gz*" extension to the compressed file name. If the compaction process fails, for example due to running out of disk space, the original file is left untouched. To restore the original file, one can use:

```
> gzip -d Makefile.gz
> ls -l Makefile*
-rw-r--r-- 1 user001 user001 361 Mar 24 17:55 Makefile
```

For convenience, the same effect may be achieved using the "*gunzip*" program (without the "-d" option).

Linux provides other file compression utilities that function in a similar fashion to *gzip*, such as *bzip2*, *lzma*, *7z*, and others. While they may achieve better data compression ratios, the compute time required to process the input files may be substantially longer.

tar (archive files)

The *tar* program has a long tradition as the primary file archiving tool for Unix. Its three primary invocation formats are:

tar —c —f archive options path...

tar —x —f archive options

tar —t —f *archive*

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 643

The first creates an archive containing all file system objects pointed to by paths (which may be files and directories). For any *path* identifying a directory, its content will be archived recursively. The options may specify the compression algorithm to be used: "-z" for *gzip*, "-j" for *bzip2*, "-J" for *xz*, and "--lzma" for *lzma*. Other useful options include verbose output "-v" and preservation of original permissions "-p".

The second form extracts the contents of the archive to the current working directory or location specified in the "-C *directory*" option. The decompression algorithm does not need to be specified, as it is automatically determined through examination of archive content. Finally, the third instance lists the contents of the specified archive.

Example:

```
> ls -l src
total 12
-rw-r--r-- 1 user001 user001  361 Mar 27 14:06 Makefile
-rw-r--r-- 1 user001 user001  491 Mar 27 14:06 heat.c
drwxrwxr-x 2 user001 user001 4096 Mar 24 21:51 other
> tar -c -f sources.tar.gz -z src
> ls -l sources.tar.gz
-rw-rw-r-- 1 user001 user001 540 Mar 27 14:08 sources.tar.gz
> tar -t -f sources.tar.gz
src/
src/other/
src/other/heat2.c
src/heat.c
src/Makefile
```

Glossary

Absolute Time in Pregroove (ATIP) An additional metadata segment that guides the process of data storage on recordable and rewritable optical media.
Abstract Device Interface for I/O (ADIO) A device-independent layer providing I/O functionality in MPI.
Accelerated Processing Unit (APU), formerly Fusion A processor architecture developed by AMD that combines conventional CPU cores and GPU logic on a single die, sharing external memory.
Accelerator A special-purpose hardware device used to speed up the execution of specific tasks.
Access Control List (ACL) Implementation of a fine-grain access control to file system entities.
Accumulator A dedicated processor register used for operand and result storage in ALU operations.
Adaptive Mesh Refinement (AMR) A numerical method employing multiple-resolution meshes adaptively in a simulation to improve efficiency and reduce the memory requirements for a simulation.
Advanced RISC Machine (ARM) A family of computer processors with fewer transistors but also lower power consumption and lower cooling requirements. RISC stands for "reduced instruction set computing."
American Standard Code for Information Interchange (ASCII) One of the most widespread character encoding standards, comprising 128 characters including all letters of the English alphabet.
AND Binary logic function that evaluates to one only when all its inputs are ones.
Andrew File System (AFS) A distributed file system developed at Carnegie Mellon University.
Antialiasing A signal processing technique that minimizes distortions due to artifacts outside the sampling band.
Apple Filing Protocol (AFP) A proprietary remote file access protocol developed by Apple; formerly AppleTalk Filing Protocol.
Application-Specific Integrated Circuits (ASICs) An integrated circuit designed for a specific application using predefined gates.
Arithmetic Logic Unit (ALU) The circuit that performs digital operations on integer numbers and logic values. Its counterpart for floating-point operations is the FPU.
ATA over Ethernet (AoE) A simple protocol for accessing block storage devices over Ethernet networks.
Automatically Tuned Linear Algebra Software Project (ATLAS) A project providing a BLAS implementation that is automatically tuned for performance.
Backfill Scheduling A job scheduling strategy that avoids starving the lower-priority jobs by scheduling them ahead of higher-priority jobs provided this will not delay the execution of the latter, effectively "filling back" the voids in the time—resource scheduling graph.
Ball Grid Array (BGA) A common high-density chip package type consisting of a grid of solder balls attached to a flat case.
Basic Linear Algebra Subprograms (BLAS) A standard interface to vector, matrix—vector, and matrix—matrix routines that have been optimized for various computer architectures.
Batch Processing A processing mode in which multiple, possibly parallel, compute jobs are executed without the involvement of a user; the opposite of interactive processing.
Binary-Coded Decimal (BCD) (a) Number encoding in which each decimal digit occupies a four-bit field; (b) several nonstandard encodings of upper-case letters, digits, and special codes using six-bit characters.
Bit Block Transfers (BitBLT) A set of memory copy and bit-wise compositing operations used in computer graphics and video processing.
Bit, "Binary Digit" The smallest unit of information used by most digital computers assuming one of two values, typically "0" or "1."
Block Smallest granularity of data used in transfers to and from some device types, particularly mass storage.
Blu-ray Disc (BD) Optical storage technology developed to support data volumes and transfer rates required by high-definition video.
Blu-ray Disc XL (BDXL) Blu-ray disc specification update introducing high-capacity (up to 128 GB per disc) media.
Bottleneck An execution hotspot that negatively impacts an application's performance.
Branch Prediction A hardware mechanism (frequently in combination with software support) used to determine with a high level of probability whether a conditional branch is taken or not.
Buffered File I/O An intermediate file access layer supported by the C library, often resulting in performance advantages.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 645

Burst Buffer A high-bandwidth storage device capable of quickly storing moderate amounts of data and acting as an I/O buffer between compute nodes and (slower) secondary storage.
Byte The smallest unit of addressable memory in computers, commonly comprising eight bits.
Cache A component of CPU architecture that stores a subset of main memory contents providing lower access latency and higher data bandwidth.
Cell Broadband Engine (CBE) or Cell Processor A heterogeneous multicore processor based on Power architecture and developed by Sony, Toshiba, and IBM for embedded applications.
Central Processing Unit (CPU) A primary hardware device performing code execution and data processing in a computer system; a processor.
Checkpointing The process of saving the necessary data from a running application to allow later resumption of the application in the event of system failure or to work around wallclock time execution limitations on a supercomputer.
Collaboration of Oak Ridge, Argonne, and Livermore (CORAL) A joint procurement of supercomputing resources between two key US Department of Energy National Laboratories.
Common Internet File System (CIFS) A variant of SMB protocol for remote file access.
Compact Disc (CD) An optical storage technology on 120 mm discs developed by Sony and Philips and originally used to store digital audio.
Compact Disc Read-Only Memory (CD-ROM) A variant of a CD dedicated to data storage.
Compact Disc Recordable (CD-R) An optical storage technology based on the CD format that permits one-time writing of user-defined data to the medium.
Compact Disc Rewritable (CD-RW) A variant of CD technology permitting multiple updates of medium contents.
Complementary Metal-Oxide Semiconductor (CMOS) The currently dominant technology used to fabricate integrated logic circuits.
Complex Instruction Set Computer (CISC) A type of processor architecture supporting instructions that consist of multiple low-level operations or support complex addressing modes; the opposite of RISC.
Compute Unified Device Architecture (CUDA) Nvidia's application programming interface for parallel computing on graphics processing units.
Conjugate Gradient (CG) A Krylov subspace iterative solver used for solving positive definite systems of equations.
Coprocessor A dedicated circuit accelerating a specific kind of computation.
Cross-Interleaved Reed-Solomon Code (CIRC Code) An error-detecting and error-correcting code with good spatial efficiency and well suited to correcting random and burst errors. It is used to protect the information stored on some optical media.
Cycles per Instruction (CPI) A performance metric specifying average number of processor cycles for each instruction performed.
Cylinder One of the physical address components used to locate data blocks on a hard-disk drive and identifying set of tracks equidistant from the spindle.
Daemon A process executing in the background and performing specific services.
Data Writes per Day (DWPD) Metric used to assess the endurance of SSDs due to a finite number of flash rewrites and equal to the number of full device capacity rewrites performed per day over the warranty period.
Debugger A tool to assist the programmer in stepping through a code in execution and examining program state.
Degrees of Freedom per Second (DOFS) The output metric for the HPGMG benchmark.
Department of Energy (DOE) The United States agency tasked with nuclear stockpile stewardship and research in science.
Die A semiconductor substrate for integrated circuit implementation.
Digital Audio Tape (DAT) A digital storage technology using magnetic tapes originally developed for digital audio recording by Sony.
Digital Data Storage (DDS) Magnetic-tape-based digital storage technology, now obsolete.
Digital Linear Tape (DLT) Digital storage technology using a magnetic tape format developed by Digital Equipment Corporation, no longer manufactured.
Digital Signal Processing (DSP) A computing technique used to extract features of, modify, or generate sampled signal values, frequently through the use of specialized hardware.
Digital Versatile Disc or Digital Video Disc (DVD) Digital storage technology involving 120 mm optical discs with increased capacity and data transfer rates compared to a CD.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 646

Digital Versatile Disc Random Access Memory (DVD-RAM) A storage technology permitting a large number of rewrites of compatible DVD media; incompatible with either DVD-R and DVD+R.

Digital Versatile Disc Recordable (DVD-R and DVD+R) A one-time recordable version of DVD; DVD-R and DVD+R denote incompatible formats of similar technology.

Digital Versatile Disc Rewritable (DVD-RW and DVD+RW) A version of DVD storage whose contents may be updated multiple times; DVD-RW and DVD+RW are incompatible formats of similar technology.

Direct Memory Access (DMA) A hardware mechanism in computers allowing memory access by system devices without interaction with the CPU.

Directory A unit of content organization within a file system which functions as a container for other directories, files, and file system entities.

Disc at Once (DAO) A recording mode in optical storage in which all data is written to a medium in a single operation.

Diskless Node A type of compute node that does not include secondary storage devices.

Distributed Lock Manager (DLM) Implementation of an algorithm for coordinating accesses to shared resources in a distributed computer system.

Dual In-Line Package (DIP) A type of case used to package integrated circuits with a low pin count.

Dynamic Random Access Memory (DRAM) A high-density variant of random access memory that requires periodic refreshing of its contents.

Eigenvalue Solvers for Petaflop Applications (ELPA) An HPC library for computing the eigenvalues and eigenvectors of Hermitian matrices.

Eight-to-Fourteen Modulation (EFM) A run-length-limited encoding technique frequently used to store data on optical media such as CDs.

Elastic Computing A type of processing in which the footprint of utilized resources may significantly vary over time.

Electrically Erasable Programmable Read-Only Memory (EEPROM) A variant of semiconductor read-only memory whose contents may be electrically erased and reprogrammed.

Environment Variable A uniquely named string defined in a context (environment) of the underlying command shell and providing additional information or configuration to specific tools or applications.

Error-Correcting Code (ECC) An additional code accompanying a data segment that permits both detection and correction of data corruption; the extent of detection and correction relies on the data size and algorithm used.

Escape Opcode A predefined prefix used in assembly code causing the CPU to transfer control to a coprocessor for the duration of the next instruction.

Event-Triggered Scheduling A simple variant of scheduling in which only jobs at the front of system queues are considered for scheduling.

Exclusive Or (XOR, EXOR or EOR) A binary function evaluating to one only if the number of one-valued arguments is odd.

External Data Representation (XDR) Data serialization layer enabling interoperability between hosts using different internal data representations.

Extreme Science and Engineering Discovery Environment (XSEDE) An NSF-funded project aiming to provide coordinated and unified access to supercomputing resources, expertise, and related tools to researchers, scientists, and engineers around the world. Formerly known as Teragrid.

Failover The process of replacing failed services in a high-availability system.

Fast Fourier Transform (FFT) A transform frequently used in signal processing and solving partial differential equations.

Fibre Channel (FC) A custom high-speed network technology used to attach storage devices to servers.

Fibre Channel over Ethernet (FCoE) A protocol encapsulating Fibre Channel communication over an Ethernet network.

Fibre Channel Protocol (FCP) A protocol encapsulating SCSI communication over a Fibre Channel connection.

Fibre Connection (FICON) Mapping IBM's specific storage access protocols onto Fibre Channel, used primarily by mainframes.

Field-Effect Transistor (FET) A semiconductor device applying a field effect controlled by the potential of the gate electrode to modulate the conductance of a channel between the source and drain electrodes.

Field Programmable Gate Array (FPGA) A device whose logical functionality may be specified and reconfigured by the user at the hardware level.

File A named entity representing a collection of data in a file system.

File Access Delegation Optimization of file data operations implemented by some versions of NFS.

File Descriptor A handle, usually an integer, identifying an open file.

File Extent A contiguous storage space reserved for file data; a file fragment.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 647

File Identifier (FID) A unique file name in the Lustre file system.

File Placement Optimizer (FPO) A feature of GPFS utilized to process "big data" workloads.

File System A high-level system for organizing and accessing data written to persistent storage devices exposing a relevant user interface and supported by the operating system.

First Come, First Serve (FIFO) A processing or data ordering structure in which individual entries are stored and processed in order of arrival; a queue.

Floating-Point Operations per Second (Flops) The output metric for the HPL and HPCG benchmarks.

Floating Gate MOS (FGMOS) Transistor A variant of MOSFET with an additional gate buried within the oxide layer which permits trapping of charge; a storage element in some nonvolatile memories.

Floating-Point Unit (FPU) A dedicated circuit performing floating-point arithmetic; a common part of CPUs and GPUs.

Gang Scheduling A scheduling strategy that groups a number of jobs, processes, or threads with similar resource requirements for the purpose of concurrent execution, allowing low latency communication between them or coordinated access to shared resources. In the Slurm Workload Manager, gang scheduling grants only one job in a gang the exclusive access to shared resources and cyclically preempts it at a timeslice boundary to enable execution of other gang members.

Gang-Partitioned (GP) Mode The initial mode of workload parallelization in OpenACC.

General Parallel File System (GPFS) A proprietary parallel file system developed by IBM, recently rebranded as IBM Spectrum Scale.

Generalized Minimum Residual Method (GMRES) A Krylov subspace iterative solver for solving general sparse systems of equations.

Generic Security Service Application Program Interface (GSS-API) A programming interface to security services standardized by the Internet Engineering Task Force (IETF).

Giant Magnetoresistance (GMR) A quantum-mechanical phenomenon in layered ferromagnetic and weakly magnetic materials providing the basis for construction of read—write heads in modern hard-disk drives.

Gigabit Ethernet (GigE) An implementation of an Ethernet network capable of a peak data rate of 1 billion bits per second.

GNU Debugger (GDB) An open source tool to assist the programmer in stepping through a code in execution.

GNU General Public License (GNU GPL) A free software license with distribution terms defined by the Free Software Foundation.

GNU Scientific Library (GSL) A library which provides a wide array of linear algebra routines, including an interface to BLAS for C and C++.

Gperftools A popular open-source code profiling and memory allocator package originally developed by Google.

Graphical User Interface (GUI) A type of interface permitting specification of input parameters and interaction with application execution through graphics-based (instead of text-only) dialogs.

Graphics Core Next (GCN) A GPU microarchitecture developed by AMD and used in its current line of products.

Graphics Processing Unit (GPU) A specialized device accelerating computations related to image or video generation.

Hadoop Distributed File System (HDFS) A file system in Hadoop which enables distributed file access across many linked storage devices.

Hamming Codes A family of error-correcting codes capable of correcting single bit errors and detecting single or double (extended hamming code) bit errors with optimal spatial overhead.

Hard Disk Drive (HDD) A storage device technology utilizing rigid, spinning, magnetic platters as media to store information.

Heap A segment of an application's memory space that hosts dynamically allocated storage.

Heterogeneous System Architecture (HSA) A set of specifications maintained by the HSA Foundation that simplifies the management and programming of heterogeneous devices sharing memory resources by providing unified architecture, API, and language support.

Hierarchical Data Format (HDF5) A library for self-describing portable data output frequently used in HPC applications.

High-Bandwidth Memory (HBM) A memory technology providing high data bandwidths utilizing a large number of interface pins and three-dimensional die stacking.

High-Density Complementary Metal-Oxide-Semiconductor (HCMOS) An older description of a CMOS process variant used to manufacture circuits with high transistor counts.

High Performance Computing (HPC) A parallel computing mode involving the use of supercomputers.

High Performance Conjugate Gradients (HPCG) A benchmark complementing the HPL benchmark which explores memory and data access patterns that are not well represented by HPL.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 648

High Performance Linpack (HPL) The third iteration of the Linpack benchmark, used for the Top 500 supercomputer ranking list.
Highly Scalable Preconditioner (HYPRE) A library developed at Lawrence Livermore National Laboratory which provides a set of highly scalable preconditioners for sparse linear system solves.
High-Throughput Computing A parallel computing strategy in which a large number of loosely coupled tasks is executing on distributed execution resources.
Hotspot In performance analysis, a part of code dominating the program execution time.
HyperSCSI A protocol implementing SCSI communication over an Ethernet network.
Hyperthread Intel's variant of multithreading in which two threads may coexecute on a single CPU core.
IBM Spectrum Scale The current name of the IBM General Parallel File System.
IEEE754 An IEEE standard defining the format of floating-point numbers.
InfiniBand Architecture (IBA) A high-speed interconnect technology found in many current HPC cluster installations.
Inode An internal data structure in a Unix-compatible OS kernel containing low-level metadata of file system objects.
Input/Output Operations per Second (IOPS) A performance metric of storage devices specifying the number of small independent I/O requests processed by the device within a second; may be further qualified as read or write accesses, random or sequential, etc.
Instruction-Level Parallelism (ILP) A type of fine-grain parallelism due to multiple operations issued as result of instruction processing.
Instruction Mix Decomposition of a computational workload or benchmark by types of instructions it executes (such as ALU, branches, memory access, etc.).
Instruction Set Architecture (ISA) A description of computer architecture based on a command set it can execute.
Instrumentation A program modification that permits extraction of specific performance data or other execution-related details.
Interactive Processing A processing mode that grants the user control over job execution, frequently used to facilitate debugging of applications.
Internet Fibre Channel Protocol (iFCP) A communication protocol enabling Fibre Channel connectivity over an IP network.
Internet Protocol over InfiniBand (IPoIB) The encapsulation of Internet Protocol traffic over physical InfiniBand fabric.
Internet Small Computer Systems Interface (iSCSI) A protocol forwarding SCSI commands over an IP network.
Internet2 A nonprofit technology community of US academic, government, research, and industrial partners founded in 1996, primarily known for advancing global research by offering access to high-bandwidth networks on a national scale.
iSCSI Extensions for RDMA (iSER) An extension of the Internet Small Computer System Interface enabling the use of remote direct memory access over the underlying network.
Isosurfaces Surfaces that connect data points which have the same value.
Job Array A collection of a specific number of jobs with similar properties and characteristics, managed as a single group.
Job Queue A named entity in a resource management system allowing grouping of jobs with similar characteristics and associated with a specific set of execution resources.
Job Step A meaningful part of a larger computational job; a task.
Joint Test Action Group (JTAG) A formative body and a resulting standard that defines the signaling interface and protocol for in-circuit access to the internal state of hardware devices.
Journaling File System A file system implementation in which uncommitted transactions are stored in a dedicated log, resulting in improved reliability.
Kerberos A network-enabled authentication software layer.
Knights Landing (KNL) The code name for a revision of Intel Xeon Phi architecture.
Linear Algebra Package (LAPACK) A linear algebra library that provides driver routines designed to solve complete problems such as a system of linear equations, eigenvalue problems, or singular value problems.
Linear Tape-Open (LTO) A digital storage technology using magnetic tapes and developed as an open standard by the LTO Consortium.
Link A construct supported by some file systems and used to provide alternative names (aliases) for stored objects.
Linpack A linear algebra library for solving systems of linear equations. It has been superseded by LAPACK.
Linux A popular open-source operating system kernel based on Unix.
Low Infrastructure Public Key Mechanism (LIPKEY) A credential exchange protocol implemented as a layer above SPKM.
Low Level Virtual Machine (LLVM) An open-source compiler project that has become a key component of development tools for Apple's MacOS and iOS.

Lustre Distributed Lock Manager (LDLM) The component of a Lustre file system responsible for efficient synchronization of concurrent accesses to shared files.

Lustre File System Check (LFSCK) A distributed file system check utility customized for Lustre.

Lustre Networking (LNET) The communication infrastructure in a Lustre file system.

M.2 The form factor and interface specification of internal expansion cards (primarily storage) attached through a miniaturized edge connector.

Management-Processing Element (MPE) A conventional core that provides directive functions, as opposed to a compute-processing element intended for computation.

Management Server (MGS) The component of a Lustre file system responsible for maintaining and providing configuration information.

Management Target (MGT) Storage space for MGS in a Lustre file system.

Many-Integrated Core (MIC) Architectural concept and hardware product introduced by Intel in which multiple tens of interconnected identical computing cores are embedded in a single device; currently known under the brand name Xeon Phi.

Mass Storage A class of storage capable of accommodating large amounts of data.

Massively Parallel Processor (MPP) A class of parallel computing architecture consisting of very large number of nodes connected by a network.

Matrix Template Library (MTL) A library for linear algebra operations that retains the look and feel of the original mathematical notation of linear algebra.

Mean Time Between Failures (MTBF) An estimated measure of system or device reliability equal to the average period of time between consecutive failures.

Memory Wall A mismatch between the computational throughput of a processor and the data rate a connected storage device (memory) is capable of supporting.

Message-Passing Interface (MPI) A programming interface and software stack used in supercomputing environments for communication between participating processes.

Metadata Additional attributes or information about stored data, typically used to indicate the owner, access rights, creation time, size, etc.

Metadata Server (MDS) The Lustre file system component managing namespace and metadata.

Metadata Target (MDT) The metadata storage in a Lustre file system.

Metal-Oxide Semiconductor Field-Effect Transistor (MOSFET) A variant of field-effect transistor (FET) with an insulated gate; a building block of electronic CMOS circuits.

Microcode A translation layer in processing hardware permitting implementation of higher-complexity instructions.

Microprocessor without Interlocked Pipeline Stages (MIPS) An influential RISC processor architecture originally developed at Stanford University.

Mini-Compact Disc (Mini-CD) A smaller version of a CD with a diameter of 80 mm.

Mini-SATA (mSATA) A miniaturized variant of a SATA connector utilized by small form factor storage devices.

MoM A job execution daemon in PBS.

Moore's Law An observation made by Gordon Moore of Fairchild Semiconductor stating that the number of transistors in large integrated circuits doubles approximately every 2 years.

Motion Picture Experts Group (MPEG) A standards group founded by ISO and IEC tasked with creating specifications for compressed digital video and audio encoding; the standards names include "MPEG-" followed by a numerical or alphabetic suffix (such as MPEG-2).

Mount Point Directory under which the contents of another file system is exposed in a process called "mounting."

MPI+X The concept of using coarse-grained MPI processes to span an entire node but allowing the efficiencies of shared-memory hardware to be exploited with the assistance of an additional programming interface like OpenMP working in cooperation with MPI.

Multi-Chip Module (MCM) A type of electronic device assembly and packaging combining several dies on a common carrier.

Multilevel Cell (MLC) Nonvolatile storage organization in which each storage cell of a device contains two bits of information.

Multiple-Mount Protection The aspect of failover management in a Lustre file system preventing simultaneous mounts on different nodes.

Multithreading A parallel execution paradigm employing multiple control flow contexts (threads) sharing an address space.

Myrinet High performance network developed by Myricom and deployed as cluster interconnect.

National Aeronautics and Space Administration (NASA) A US government agency that manages and directs the civilian space program and is a frequent driver of high performance computing applications.
National Television System Committee (NTSC) A standard defining video stream properties, color encoding, and the transmission modulation scheme for the analog television signal used in most of the Americas and some Pacific territories.
National University of Defense Technology (NUDT) The top military academy and defense research university in Changsha, Hunan, China. NUDT supports both supercomputing research and the Chinese space program.
Network Attached Storage (NAS) A shared network-connected storage pool accessible remotely through specialized protocols and software.
Network File System (NFS) A remote, shared file access service with a protocol defined by several open RFC standards and commonly used in Unix environments.
Network Interface Controller (NIC), also Network Interface Card A specialized electronic device or adapter board that allows connecting the computer to a specific network type.
Network Shared Disk (NSD) Storage abstraction in IBM Spectrum Scale (GPFS).
Noise-Predictive Maximum Likelihood (NPML) A set of digital signal processing methods used to improve the reliability of retrieved information from noisy channels or media (such as magnetic disks).
Nonuniform Memory Access (NUMA) A memory architecture in which memory access latency varies depending on the relative location of the issuing processor and targeted memory module.
Nonvolatile Memory Express (NVMe) The interface specification for attaching nonvolatile storage devices over a PCI express bus.
Nonvolatile Random Access Memory (NVRAM) A class of memory whose contents are retained after device power is turned off.
NAND Binary logic function that evaluates to zero only when all its inputs are ones.
NOR Binary logic function that evaluates to one only when all its inputs are zeroes.
NVLink A short-range communications protocol between a GPU and a CPU or multiple GPUs, developed by Nvidia.
Object Storage Server (OSS) Processes file data requests in a Lustre file system.
Object Storage Target (OST) The underlying physical storage for OSS in a Lustre file system.
Offline Storage A variant of archival storage in which access to storage media is explicitly managed by a human operator.
Open Accelerators (OpenACC) A programming model for accelerators using an approach similar to OpenMP.
Open Computing Language (OpenCL) An application programming framework providing a unified interface to execution resources, including conventional CPUs and various accelerator types.
Open Multiprocessing (OpenMP) A compiler-supported programming environment enabling application parallelization on shared-memory multiprocessors.
OpenFabrics Enterprise Distribution (OFED) A set of software stack components and protocols developed and distributed by OpenFabrics Alliance in support of InfiniBand technology.
Operating System (OS) A system software layer that allocates and manages hardware resources, enforces resource protection, provides standardized services, and schedules execution of applications.
OR Binary logic function that evaluates to zero only when all its inputs are zeroes.
Overhead An additional amount of work required to manage a computation.
Packet Writing A method of contents modification on recordable or rewritable optical media that permits addition and deletion of files and directories at any time.
Page A unit of memory organization and address translation, ranging from a few KB to a few GB.
Parallel Boost Graph Library (PBGL) A library for high performance graph algorithms.
Parallel File System A file system optimized for concurrent access to data objects.
Parallel NFS (pNFS) An extension to NFS supporting parallel access to shared files.
Partial Response Maximum Likelihood (PRML) A set of algorithms in signal theory used to increase the reliability of information retrieved from weak or interfering signals.
Path The name identifying a specific entity or object in a file system.
Perf (on occasion perf_events, perf tools or Performance Counters for Linux, PCL) A performance-monitoring and event-tracing tool available for Linux systems.
Performance Application Programming Interface (PAPI) A library which provides tools for performance measurement and portable access to hardware performance counters.
Peripheral Component Interconnect (PCI) A parallel expansion bus standard.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 651

Peripheral Component Interconnect Express (PCIe or PCI Express) A serial expansion bus standard with a control protocol derived from and extending that of PCI.

Perpendicular Recording A method of storing information on a magnetic medium that results in increased bit density compared to more traditional horizontal recording.

Picture Element (Pixel) The smallest, indivisible element of a digital image.

Pin Grid Array (PGA) An integrated circuit enclosure placing I/O leads on the bottom of ceramic or plastic case.

Plastic Leaded Chip Carrier (PLCC) A type of enclosure used to house integrated circuits with leads arranged along the four sides of a rectangular case.

Portable Batch System (PBS) A common cluster-oriented resource management system developed and maintained by Altair Engineering; recently open sourced.

Portable, Extensible Toolkit for Scientific Computation (PETSc) A suite of data structures and routines for solving partial differential equations on distributed-memory architectures.

Portable Operating System Interface (POSIX) A collection of IEEE standards specifying operating environment, programming interfaces, and interaction and management of executing entities for compatibility and interoperability across variants of the Unix operating system.

Preemption A scheduler feature allowing it to interrupt and suspend an already running lower-priority task to start the execution of a higher-priority task.

Prefetch A mechanism reducing data access latency by initiating data transfer ahead of actual data use.

Primary Storage The top level of storage hierarchy, including CPU registers, caches, and main memory.

Printed Circuit Boards (PCB) An insulated board providing mechanical support for interconnected electrical components.

Process Identifier (PID) A number used by the operating system to identify an active process.

Processing Element A primitive hardware computing unit; one of many replicated components of a processing array or a vector unit.

Profiling A performance analysis technique that measures the dynamic properties of program execution.

Programmed Input/Output (PIO) A method of data transfer between computer memory and a system device explicitly performed by the CPU.

Pseudo File System A data structure or service exposing an access interface compatible with a file system API.

Raster Operation (ROP) An operation executed during one of the final steps in computer image rendering, generating the actual displayed pixel value.

Reduced Instruction Set Computer (RISC) A processor architecture paradigm emphasizing ISAs with fewer, simpler, and more generic instructions rather than complex ones.

Redundant Array of Independent Disks, formerly Redundant Array of Inexpensive Disks (RAID) A form of aggregated storage incorporating multiple HDDs or SSDs capable of tolerating a limited number of device failures.

Remote Direct Memory Access A low-overhead data transfer technique between the memories of two machines that avoids direct involvement of their processors.

Remote Procedure Call (RPC) A distributed computing paradigm in which the client node supplies input arguments and requests the invocation of a function using these arguments on a remote server in a specific application's address space.

Request for Comments (RFC) A publication body maintained by the IETF and Internet Society, and used as a forum for internet standard development.

Request Replay Cache A data structure in some NFS implementations used to avoid duplicate request execution.

Resource Management A collection of methodologies, algorithms, and tools supporting efficient allocation of computing resources to executable tasks.

Restart At designated points during the execution of an application on a supercomputer the data necessary to allow later resumption of the application at that point in the execution can be output and saved. This data is called a checkpoint, and the resumption of application execution is called restart.

Round-Robin A task scheduling or data distribution method in which tasks or data units are repetitively assigned to resources in the same predefined order.

Scalable Library for Eigenvalue Problem Computations (SLEPc) An extension of PETSc for solving very large sparse eigenvalue problems.

SCSI RDMA Protocol or SCSI Remote Protocol (SRP) A protocol leveraging the use of remote direct memory access for SCSI commands over supporting networks such as InfiniBand or 10 Gbps Ethernet.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 652

Secondary Storage The second level of storage hierarchy, incorporating high-bandwidth mass-storage devices for persistent preservation of data.

Sector A unit of data access on storage devices; a block.

Self-Monitoring, Analysis, and Reporting Technology (SMART) A self-contained, built-in monitoring system analyzing the health status of storage devices such as HDDs and SSDs.

Serial Advanced Technology Attachment (SATA) A high-speed serial interface bus used to connect storage devices to motherboards and I/O expansion cards.

Server Message Block (SMB) A proprietary protocol with currently open specifications for remote file, printer, and hardware port access originated in the Microsoft Windows environment.

Service Unit (SU) A metric for charging supercomputer time against a user account. While defined locally for each supercomputer, it is generally considered the wall time in hours multiplied by the number of cores used for a simulation.

Setgid Analogous to "setuid," but applied to user groups.

Setuid A flag associated with an executable file changing the effective program's ownership even when executed by ordinary users; typically used to elevate the privilege level.

Shader A replicated processing component in a GPU that supports a number of bit-wise and arithmetic operations.

Shell An interface facilitating the execution of operating system commands and user programs as well as visualization of their output.

Simple Public Key GSS-API Mechanism (SPKM) An authentication protocol defined by RFC2025.

Single Instruction, Multiple Data (SIMD) An element of Flynn's taxonomy for achieving parallelism where several processing units perform the exact same operation simultaneously on multiple data inputs.

Single Program, Multiple Data (SPMD) An element of Flynn's taxonomy for achieving parallelism, and the most common style of parallel programming for distributed-memory architectures.

Single-Level Cell (SLC) Nonvolatile storage organization in which each storage cell of a device contains exactly one bit of information.

Slurm Partition Slurm's equivalent of a job queue.

Slurm Workload Manager, "Slurm" A popular open-source resource management suite for cluster computers, originally an acronym of "Simple Linux Utility for Resource Management."

Small Computer System Interface (SCSI) A standard family describing electrical and mechanical interfaces, communication protocols, and supported device functions for various peripherals such as HDDs, tape drives, scanners, and others.

Small-Scale Integration (SSI) A scale of integrated circuit minimization placing tens of transistors on a single die.

Socket A physical connector on the motherboard accommodating a CPU or a representation of physical resources provided by a single CPU package.

Solid-State Drive or Solid-State Disk (SSD) A storage device technology leveraging solid-state devices (such as flash memory) for persistent data storage and thus containing no moving components.

Starvation, Latency, Overhead, Contention, Energy, Resilience (SLOWER) Sources of performance degradation.

Stateless Protocol A communication protocol in which neither client nor server is required to retain session-related information.

Static Random Access Memory (SRAM) A variant of memory technology with the fastest access time but a higher unit cost and lower storage density than DRAM.

Sticky Bit A flag associated with files or directories that restricts when they may be deleted.

Storage Area Network (SAN) A storage virtualization layer using a network to provide block-level accessibility to remote storage devices.

Stream Multiprocessor (SM) A component of Nvidia GPU architecture consisting of multiple shader units with related infrastructure that execute concurrent compute threads.

Streamlines Streamlines take a vector field as input and show curves that are tangent to the vector field.

Stripe In distributed storage, the smallest sequence (or its size) of data blocks spanning all devices in the array.

Supercomputer A computing system exhibiting high-end performance capabilities and resource capacities within practical constraints of technology, cost, power, and reliability.

Superconducting Josephson Junction Logic (Superconducting JJ) Two superconductors coupled together across a thin insulating barrier or nonsuperconducting metal.

Symmetric Multiprocessor (SMP) The most common type of shared-memory compute node.

System on Chip (SoC) An integrated circuit containing multiple components of a computing system (CPU, memory, signal converters, graphics processors, analog functions, etc.) on a single die.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 653

TaihuLight Currently the fastest supercomputer in the world, located in China.

Tarball A file that contains a group of archived files with the extension .tar. It is often compressed using gzip, resulting in the filename extension .tar.gz.

TeraBytes Written (TBW) A metric estimating the maximum aggregate volume of data that may be written to a storage device without causing its failure or data loss.

Tertiary Storage A storage hierarchy level maintaining large amounts of data, frequently supporting automated media changes.

Texture Element (Texel) In computer graphics, a basic unit of texture.

Time-Limited Error Recovery (TLER) A property of a storage device that bounds the time required to process an internal error, making is suitable for use with RAID controllers.

TOP500 List A ranked listing of the world's fastest 500 supercomputers, updated twice a year.

Track at Once (TAO) A recording mode used in optical storage in which data may be added to a disc in several sessions.

Translation Lookaside Buffer (TLB) A critical component of modern CPUs, accelerating virtual to physical address translation.

Transmission Control Protocol (TCP) One of the commonly used internet protocols providing reliable, connection-oriented data transfer.

Traverse Edges per Second (TEPS) The output metric for the Graph500 benchmark.

Triple-Level Cell (TLC) Nonvolatile storage organization in which each storage cell of a device contains three bits of information.

Tuning and Analysis Toolkit (TAU) An open-source performance measurement, analysis, and visualization suite developed by the University of Oregon.

Uniform Memory Access (UMA) UMA shared memory is a memory architecture in which memory access latency does not vary depending on the relative location of the issuing processor and targeted memory module.

Universal Standard Bus (USB) A short-range peripheral interconnect standard.

Unix A family of multiuser operating systems descended from the original Unix developed at AT&T Bell Laboratories.

User Datagram Protocol (UDP) A simple connectionless communication protocol supporting messaging over the internet.

VampirTrace A fine-grain trace collection tool commonly used to profile MPI and OpenMP applications.

Vector-Partitioned (VP) Mode The finest grain of workload parallelization in OpenACC that uses the SIMD capabilities of the accelerator.

Very Large-Scale Integration (VLSI) The currently highest level of integrated circuit miniaturization, placing several thousands to billions of transistors on a single die.

Vienna Ab Initio Simulation Package (VASP) A widely used density functional theory toolkit for HPC systems.

Virtual Address Extension (VAX) An ISA, and a family of microcomputers based on it, developed by Digital Equipment Corporation in the 1970s.

Virtual File System (VFS) A system-independent abstraction of a file system.

Virtual Memory A memory abstraction and management technique allowing mapping of regions of an address space to different types of underlying physical storage devices.

Virtual Node or Vnode A system-independent representation of an inode in a virtual file system.

Visualization Toolkit (VTK) A visualization library that provides hundreds of visualization algorithms.

Von Neumann Bottleneck See "memory wall."

Wide Area Network (WAN) A communication network that spans large distance.

Worker-Partitioned (WP) Mode A method of workload parallelization supported by OpenACC that uses multiple workers per gang.

Xeon Phi See "many integrated core."

Yet Another Resource Negotiator (YARN) A central resource manager used in Hadoop.

Z-Buffer A data structure used in computer graphics to provide the accurate depth coordinate for each rendered pixel.

ZFS (initially Zettabyte File System) An advanced-featured file system developed by Sun Microsystems and currently owned by Oracle.

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 654

Index

'*Note:* Page numbers followed by "f" indicate figures, "t" indicate tables, and "b" indicate boxes.'

A

Accelerator architecture, 451–453
  evolution of graphics processing unit functionality, 466–471, 468f–469f, 470t
  graphics processing units, 464–466, 465f
  heterogeneous system architecture, 477–478
  historic perspective, 454–463
    accelerators in processor I/O space, 461
    accelerators with industry-standard interfaces, 462–463
    coprocessors, 456–461
      Intel 8087, 457–459
      Motorola MC68881, 459–461
  modern graphics processing unit architecture, 471–477, 472t
    compute architecture, 471–474, 474f
    interconnects, 475–476
    memory implementation, 474–475
    programming environment, 476–477
  outcomes, 480
Accelerators
  with industry-standard interfaces, 462–463
  in processor I/O space, 461
Actor synchronization, 615
Advection equation using finite difference, 295–297, 296f–298f
Aggregated storage, 534–544
  network attached storage, 543–544
  redundant array of independent disks, 532f, 534–541
    hybrid redundant array of independent disks variants, 539–541
    RAID 0: striping, 534–535
    RAID 1: mirroring, 535
    RAID 2: bit-level striping with hamming code, 536
    RAID 3: byte-level striping with dedicated parity, 536–537
    RAID 4: block-level striping with dedicated parity, 537
    RAID 5: block-level striping with single distributed parity, 538
    RAID 6: block-level striping with dual distributed parity, 539
  storage area networks, 541–542
  tertiary storage, 544, 544f, 545t
Allgather, 272–274
Alltoall, 277–278, 277f
Amdahl's law, 70–73
Amdahl's law plus, 196–199, 197f–199f
Anatomy of supercomputer, 14–16
  computer performance, 16–21
  peak performance, 17–18
  performance, 16
  performance degradation, 19–20
  performance improvement, 20–21
  scaling, 18–19
  sustained performance, 18
Antikythera mechanism, 53f
Application profiling, significance of, 390–391
Application programming, 4, 8–9
Application-level checkpointing, 598–601
Architecture overview, essential SLURM, 147, 147f
Arithmetic logic unit (ALU), 63–64, 70
Arithmetic on variables, Linux, 654
Asynchronous multitasking, 613–615
  actor synchronization, 615
  global address space, 614–615
  message-driven computation, 613–614
  multithreaded, 613
  runtime system software, 615
Atomics, 504–505
Automated calculators through mechanical technologies, 22–23

B

Backus, John, 317
Barrier directive, OpenMP programming model, 243
Basic linear algebra subprograms (BLAS), 317–322
  Level 1 rotation operations, 320t
  Level 1 vector operations, 320t–321t
  Levels 2 and 3, 321t–322t
  routines, precision prefixes used by, 319t
Basic methods of use, commodity clusters, 104–113
  compilers and compiling, 112–113
  logging on, 104–105
  package configuration and building, 110–111
  running applications, 113
  user space and directory system, 105–110
Basic two-input logic gates, 56f
Benchmarking
  benchmarks used in HPC community, 118t
  Graph500, 132–135
  highly parallel computing linpack, 120–123
  high performance conjugate gradients, 126–128
  HPC challenge benchmark suite, 123–124

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 655

Benchmarking (*Continued*)
key properties of, 117–120
miniapplications as benchmarks, 135–138
NAS parallel benchmarks, 130–131
nonproprietary, 119t
outcomes, 138–139
overview, 115–117
standard HPC community benchmarks, 120
top-performing supercomputer, 119t
Beowulf class of parallel computer architecture, 77f
Beowulf cluster project, 91–93
Bit-level striping with hamming code (RAID 2), 536
Block-level striping
with dedicated parity (RAID 4), 537
with dual distributed parity (RAID 6), 539
with single distributed parity (RAID 5), 538
Branch prediction, symmetric multiprocessor architecture, 201–202
Breadth first search, parallel algorithms, 306–310, 307f–309f
Brief history of supercomputing, 21–38
automated calculators through mechanical technologies, 22–23
communicating sequential processors and very large scale integration, 34–37
instruction-level parallelism, 29–30
multicore petaflops, 37
neodigital age and beyond Moore's law, 37–38
single-instruction multiple data array, 33
vector processing and integration, 30–33
von Neumann architecture in vacuum tubes, 24–29
Broadcast, 268–269
Buffer flush, 560
Buffered File I/O, 559–562
buffer flush, 560
conversion between streams and file descriptors, 561–562
file open and close, 559
offset update and query, 560
sequential data access, 560
Bush, Vannevar, 22–23
Byte-level striping with dedicated parity (RAID 3), 536–537

C

C++ AMP, 486
Cannon's algorithm, 301–303, 301f–307f
Catalyzing fraud detection, 10
CDC-7600, 318f
Cellular automata, 618–619
Checkpointing, 591
application-level, 598–601
outcomes, 602
system-level, 592–597
Checkpoint/restart, 47–48
Clearspeed Advance e710 accelerator board, 463f
Climate change, understanding of, 12–14
CMOS, 55
Collective data movement, 265–267, 265f–267f
Command substitution, Linux, 654
Command-line utilities, Linux, 659–663
data compression and archiving, 662–663
process management, 661–662
text tools, 659–661
Commands, SLURM scheduling, 151–166
sacct, 164–165
salloc, 160–161
sbatch, 161–162
scancel, 163–164
sinfo, 165–166
squeue, 162–163
srun, 151–160
Commercial parallel debuggers, 431–432
Commercial systems summary, commodity clusters, 95
Commodity clusters
basic methods of use, 104–113
compilers and compiling, 112–113
logging on, 104–105
package configuration and building, 110–111
running applications, 113
user space and directory system, 105–110
Beowulf cluster project, 91–93
cluster elements, 85–86
definition of, 84
hardware architecture, 93–95
commercial systems summary, 95
node, 93–94
secondary storage, 95
system area networks, 94–95
history, 88–90
HPC architecture, 77–78
impact on top 500 list, 86–87
motivation and justification for clusters, 84–85
outcomes, 113–114
overview, 84–90
programming interfaces, 97–98
high performance computing programming languages, 97
parallel programming modalities, 97–98
software environment, 98–104
operating systems, 98–99
resource management, 99–100
debugger, 101
performance profiling, 101
visualization, 101–104
Communicating sequential processors and, 34–37

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 656

Communication collectives, 265–278
allgather, 272–274
alltoall, 277–278, 277f
broadcast, 268–269
collective data movement, 265–267, 265f–267f
gather, 271–272
reduction operations, 274–276, 274t
scatter, 269–271
Communicators, 255–258
example, 257–258
rank, 256–257
size, 256
Compilation, Linux, 656–659
Compiler flags for debugging, 439
Compilers and compiling, commodity clusters, 112–113
Compute architecture, 471–474, 474f
Compute unified device architecture, 485
Computer performance, 16–21
Concurrent applications, SLURM job scripting, 167–169
Connection Machine 2 (CM-2), 454–463
Connection Machine 5 (CM-5), 252f
Control Data Corporation (CDC) 6600, 51–55
Control flow, Linux, 654–656
Conversion between streams and file descriptors, 561–562
Coprocessors, 456–461
Intel 8087, 457–459
Motorola MC68881, 459–461
Cray, Seymour, 30–33
Cray-1 Supercomputer, 32f
Critical synchronization directive, OpenMP programming model, 242

D

Data access with explicit offset, 556
Data compression and archiving, Linux, 662–663
Data management, 497–501
Data reuse and locality, symmetric multiprocessor architecture, 204–205
Dataflow, 617–618
Deadlock, 434–439
Debugger, commodity clusters, 101
Debugging, 421–422
compiler flags for debugging, 439
debugging MPI example, deadlock, 434–439
outcomes, 445
system monitors to aid debugging, 441–445
tools, 423–432
commercial parallel debuggers, 431–432
GNU debugger, 423–430
back trace, 427–428, 428f–429f
break points, 424–425, 425f, 425t, 426f
GDB cheat sheet, 430
setting a variable, 428–430, 430f
threads, 430, 431f–432f
watch points and catch points, 425–426, 427f
Valgrind, 430–431, 433t
unprotected shared variable assessment, 433–434, 436f
Digital logic, HPC architecture, 55–58
Directives, OpenMP programming model, 230–231
Discovering oil and gas, 10
Distributed computation, MapReduce, 584
Distributing oil and gas, 10
Divide and conquer, parallel algorithms, 287–291, 288f–291f
Domain-specific languages for linear algebra, 329, 330f–332f
Dongarra, Jack, 116b
Dynamic random access memory (DRAM), 45, 61–62

E

Earth Simulator (ES), 64b
Efficiency, HPC architecture, 46
Eigenvalue SoLvers for Petaflop-applications, 328
Elastic computing, SLURM scheduling, 150
Electronic numerical integrator and computer (ENIAC), 51f
Emacs, 645–646
Embarrassingly parallel, parallel algorithms, 292–293, 293f
Enabling technology, HPC architecture, 51–62
digital logic, 55–58
memory technologies, 58–62
early memory devices, 59–61
modern memory technologies, 61–62
roles of technologies, 55
technology epochs, 51–55
Environment variables
OpenMP programming model, 229–230
SLURM job scripting, 169–171
Environment variables of interest, 185–186
Essential bash, Linux, 647–656
arithmetic on variables, 654
command substitution, 654
control flow, 654–656
input/output, 649–650
path expansion, 647–648
pipelines, 649–650
redirection, 649–650
special character handling, 649
variables, 650–653
Essential gperftools, 391–398
Essential Linux
command-line utilities, 659–663
data compression and archiving, 662–663
process management, 661–662
text tools, 659–661

Essential Linux (*Continued*)
compilation, 656–659
essential bash, 647–656
arithmetic on variables, 654
command substitution, 654
control flow, 654–656
input/output, 649–650
path expansion, 647–648
pipelines, 649–650
redirection, 649–650
special character handling, 649
variables, 650–653
files editing, 645–647
Emacs, 645–646
Gedit, 646
Kate, 646
Nano, 646
Vi, 645
logging In, 637–638
navigating the file system, 641–645
remote access, 639–640
Essential MPI, 250
communication collectives, 265–278
allgather, 272–274
alltoall, 277–278, 277f
broadcast, 268–269
collective data movement, 265–267, 265f–267f
gather, 271–272
reduction operations, 274–276, 274t
scatter, 269–271
communicators, 255–258
example, 257–258
rank, 256–257
size, 256
message-passing interface basics, 253–255
message-passing interface example, 254–255
MPI_Finalize, 254
mpi.h, 253
MPI_Init, 253
message-passing interface standards, 251–253
nonblocking point-to-point communication, 279–281
outcomes, 283
point-to-point messages, 258–262
example, 260–262
message-passing interface data types, 259, 260t
MPI Recv, 259–260
MPI send, 259
synchronization collectives, 262–265
barrier synchronization, 263, 263f
example, 264–265
overview of collective calls, 262–263
user-defined data types, 281–283
Essential OpenMP, 225–226
outcomes, 245–246
overview of OpenMP programming model, 226–231
runtime library and environment variables, 228–231
directives, 230–231
environment variables, 229–230
runtime library routines, 230
thread parallelism, 226–228
thread variables, 228
parallel threads and loops, 231–240
parallel "for", 233–238
parallel threads, 231–232
private, 232–233
sections, 239–240
reduction, 244
synchronization, 241–243
barrier directive, 243
critical synchronization directive, 242
master directive, 242–243
single directive, 243
Essential portable batch system, 172–187
PBS cheat sheet, 186–187
PBS commands, 174–183
pbsnodes, 183
qdel, 180
qstat, 180–182
qsub, 174–179
tracejob, 182–183
PBS job scripting, 184–186
environment variables of interest, 185–186
MPI jobs, 185
OpenMP jobs, 184–185
portable batch system architecture, 173–174, 173f
portable batch system overview, 172–173
Essential POSIX file interface, 554–562
buffered File I/O, 559–562
buffer flush, 560
conversion between streams and file descriptors, 561–562
file open and close, 559
offset update and query, 560
sequential data access, 560
system calls for file access, 554–558
data access with explicit offset, 556
file length adjustment, 556–557
file offset manipulation, 556
file open and close, 554–555
file status query, 557–558
sequential data access, 555–556
synchronization with storage device, 557

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 658

Essential resource management
essential portable batch system, 172–187
PBS cheat sheet, 186–187
PBS Commands, 174–183
pbsnodes, 183
qdel, 180
qstat, 180–182
qsub, 174–179
tracejob, 182–183
PBS job scripting, 184–186
environment variables of interest, 185–186
MPI jobs, 185
OpenMP jobs, 184–185
portable batch system architecture, 173–174, 173f
portable batch system overview, 172–173
essential SLURM, 146–172
architecture overview, 147, 147f
commands, 151–166
sacct, 164–165
salloc, 160–161
sbatch, 161–162
scancel, 163–164
sinfo, 165–166
squeue, 162–163
srun, 151–160
SLURM cheat sheet, 171–172
SLURM job scripting, 166–171
concurrent applications, 167–169
environment variables, 169–171
MPI scripts, 167
OpenMP scripts, 167
script components, 166–167
SLURM scheduling, 149–150
elastic computing, 150
gang scheduling, 149
generic resources, 150
high-throughput computing, 150
preemption, 149–150
trackable resources, 150
workload organization, 148, 148f
managing resources, 142–146
outcomes, 187–189
Essential SLURM, 146–172
architecture overview, 147, 147f
commands, 151–166
sacct, 164–165
salloc, 160–161
sbatch, 161–162
scancel, 163–164
sinfo, 165–166
squeue, 162–163
srun, 151–160
SLURM cheat sheet, 171–172
SLURM job scripting, 166–171
concurrent applications, 167–169
environment variables, 169–171
MPI scripts, 167
OpenMP scripts, 167
script components, 166–167
SLURM scheduling, 149–150
elastic computing, 150
gang scheduling, 149
generic resources, 150
high-throughput computing, 150
preemption, 149–150
trackable resources, 150
workload organization, 148, 148f
Ethernet, 213–214
Exascale computing, 610–612
accelerated approach, 612
capacity, 611–612
challenges to, 611
lightweight cores, 612
Execution pipeline, symmetric multiprocessor architecture, 200–201
Expanded parallel programming models, 606–607
advance in message-passing interface, 606–607
advances in OpenMP, 607
MPI+X, 607
Extended high performance computing architecture, 608–609
field programmable gate arrays, 609
lightweight architectures, 608–609
world's fastest machine, 608
External I/O interfaces, symmetric multiprocessor architecture, 213–222
JTAG, 218–219
network interface controllers, 213–215
Ethernet, 213–214
InfiniBand, 215
serial advanced technology attachment, 215–218
universal serial bus, 220–222

F

Fast Fourier transform (FFT), 628–631
Field programmable gate arrays, 609
File length adjustment, 556–557
File management, 350
File offset manipulation, 556
File open and close, 554–555, 559
File status query, 557–558

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 659

File systems, 86
essential POSIX file interface, 554–562
buffered File I/O, 559–562
buffer flush, 560
conversion between streams and file descriptors, 561–562
file open and close, 559
offset update and query, 560
sequential data access, 560
system calls for file access, 554–558
data access with explicit offset, 556
file length adjustment, 556–557
file offset manipulation, 556
file open and close, 554–555
file status query, 557–558
sequential data access, 555–556
synchronization with storage device, 557
general parallel file system, 565–569, 566f, 566t
Lustre file system, 569–575
network file system, 562–565, 563f
outcomes, 575–576
role and function, 549–553
Files editing, Linux, 645–647
Emacs, 645–646
Gedit, 646
Kate, 646
Nano, 646
Vi, 645
Flops, 46
Flynn's taxonomy, 48–50, 50f
Fork–join model
master/worker threads, 228f
parallel algorithms, 286–287, 287f–288f
Forwarding, symmetric multiprocessor architecture, 202
Foundational visualization concepts, 364

G

Gang scheduling, SLURM scheduling, 149
Gara, Alan, 610–612
Gather, 271–272
Gedit, 646
General parallel file system, 565–569, 566f, 566t
Generic resources, SLURM scheduling, 150
Gigabit Ethernet
network interface card, 214f
switch, 214f
Global address space, 614–615
GNU debugger, 423–430
back trace, 427–428, 428f–429f
break points, 424–425, 425f, 425t, 426f
GDB cheat sheet, 430
setting a variable, 428–430, 430f
threads, 430, 431f–432f
watch points and catch points, 425–426, 427f
GNU scientific library, 326, 327f
Gnuplot, 365–368, 366f–370f
gperftools, 391–398
Graph algorithms, 329
Graph500, 132–135, 133f–134f, 134t, 135f–137f
Graphics processing units, 464–466, 465f
functionality, evolution of, 466–471, 468f–469f, 470t
modern architecture, 471–477, 472t
compute architecture, 471–474, 474f
interconnects, 475–476
memory implementation, 474–475
programming environment, 476–477
Gropp, William D., 251f
Gyrokinetic Toroidal code, 9f

H

Hadoop, 585–588
Halo exchange, parallel algorithms, 294–299, 294f
advection equation using finite difference, 295–297, 296f–298f
sparse matrix vector multiplication, 297–299, 299f–300f
Hard-disk drives, 514–520, 515f, 517f, 519t
Hardware architecture, commodity clusters, 93–95
commercial systems summary, 95
node, 93–94
secondary storage, 95
system area networks, 94–95
Hardware fault, 47–48
Heterogeneous computer structures, 78
Heterogeneous system architecture, 477–478
High performance computing
computational science research areas, 314f
disciplines, 3–9
application programming, 8–9
application programs, 4
definition, 3–4
high performance computing systems, 5–7
performance and metrics, 4–5
supercomputing problems, 7–8
programming languages, commodity clusters, 97
systems, 5–7, 7f
High performance linpack, 120–123, 122f
High performance conjugate gradients, 126–128
High throughput computing, SLURM scheduling, 150
Historic perspective, of accelerator architecture, 454–463
accelerators in processor I/O space, 461
accelerators with industry-standard interfaces, 462–463
coprocessors, 456–461
Intel 8087, 457–459
Motorola MC68881, 459–461
Hopper, Grace Brewster Murray, 421f, 423f

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 660

Host node, 86
HPC architecture
enabling technology, 51–62
digital logic, 55–58
memory technologies, 58–62
early memory devices, 59–61
modern memory technologies, 61–62
roles of technologies, 55
technology epochs, 51–55
heterogeneous computer structures, 78
key properties, 44–48
efficiency, 46
parallelism, 45–46
power, 46–47
programmability, 48
reliability, 47–48
speed, 45
multiprocessors, 73–78
commodity clusters, 77–78
massively parallel processors, 76
shared-memory multiprocessors, 74–76
outcomes, 78–79
overview, 44
parallel architecture families, Flynn's taxonomy, 48–50
single-instruction, multiple data array, 69–73
Amdahl's law, 70–73
single-instruction, multiple data architecture, 69–70
vector and pipelining, 64–68
pipeline parallelism, 65–68
vector processing, 68
von Neumann sequential processors, 62–64
HPC challenge benchmark suite, 123–124
Hybrid redundant array of independent disks variants, 539–541
Hypre, scalable linear solvers and multigrid methods, 328–329

I

IBM Blue Gene supercomputers, 610–612
ILLIAC (Illinois Automatic Computer), 454–463
Independent disks variants, hybrid redundant array of, 539–541
InfiniBand, 215
port, 216f
Input/output , Linux, 649–650
Instruction interface, 70
Instruction set architecture (ISA), 44, 54
Instruction-level parallelism, 29–30
symmetric multiprocessor architecture, 201
Integrated performance monitoring toolkits, 407–410, 410f
Intel 8087, 457–459, 457f–458f
Intel Touchstone Delta, 35–36
Interconnects, 475–476
I/O system management, 351
Isosurfaces, 364, 365f

J

JTAG, 218–219, 219f

K

Kate, 646
Kernels construct, 495–496
Key properties, HPC architecture, 44–48
efficiency, 46
parallelism, 45–46
power, 46–47
programmability, 48
reliability, 47–48
speed, 45
K-means clustering, 582–584

L

Lapack, 324, 325f
driver routines, 324t
Libraries, 313–315, 315t
graph algorithms, 329
linear algebra, 315–329, 316f
basic subprograms, 317–322
domain-specific languages for, 329, 330f–332f
eigenvalue SoLvers for Petaflop-applications, 328
GNU scientific library, 326
Hypre, scalable linear solvers and multigrid methods, 328–329
package, 324–326
portable extensible toolkit for scientific computation, 327–328
scalable library for eigenvalue problem computations, 328
scalable linear algebra, 326
supernodal LU, 326–327
mesh decomposition, 333–334, 340f
outcomes, 342–343
parallel input/output, 330–332, 335f
parallelization, 334
partial differential equations, 329
performance monitoring, 341–342
signal processing, 334–341
visualization, 334
Lightweight architectures, 608–609
Linear algebra, 315–329, 316f
basic subprograms, 317–322
domain-specific languages for, 329, 330f–332f
eigenvalue SoLvers for Petaflop-applications, 328

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 661

Linear algebra (*Continued*)
GNU scientific library, 326
Hypre, scalable linear solvers and multigrid methods, 328–329
package, 324–326
portable extensible toolkit for scientific computation, 327–328
scalable library for eigenvalue problem computations, 328
scalable linear algebra, 326
supernodal LU, 326–327
Linpack, 116b, 120–123
Local registers, 70
Logging in, Linux, 637–638
Logging on, commodity clusters, 104–105
Loop scheduling, 501–504
Lower/upper decomposition, 624–628
Lustre file system, 569–575

M

Magnetic tape, 524–528, 527f–528f, 529t
Magnetism, 60–61
Makefiles, 111
Manager–worker, parallel algorithms, 291–292, 292f
Managing oil and gas, 10
Managing resources, 142–146
Manufacturing, accelerating innovation in, 10–11
MapReduce, 579
distributed computation, 584
Hadoop, 585–588
Map and Reduce, 579–584
K-means clustering, 582–584
shared neighbors, 581–582
word count, 581
outcomes, 588–589
Market data analytics, 10
Mass storage, 509–512
aggregated storage, 534–544
network attached storage, 543–544
redundant array of independent disks, 532f, 534–541
hybrid redundant array of independent disks variants, 539–541
RAID 0: striping, 534–535
RAID 1: mirroring, 535
RAID 2: bit-level striping with hamming code, 536
RAID 3: byte-level striping with dedicated parity, 536–537
RAID 4: block-level striping with dedicated parity, 537
RAID 5: block-level striping with single distributed parity, 538
RAID 6: block-level striping with dual distributed parity, 539
storage area networks, 541–542
tertiary storage, 544, 544f, 545t
history, 512, 513f
outcomes, 545–546
storage device technology, 514–532
hard-disk drives, 514–520, 515f, 517f, 519t
magnetic tape, 524–528, 527f–528f, 529t
optical storage, 529–532, 530f
solid-state drive storage, 520–524, 521f–522f, 525t, 529t
Massively parallel processor (MPP), 55, 76f
HPC architecture, 76
Master directive, OpenMP programming model, 242–243
Matplotlib, 369–372, 371f–372f
Mellanox IB cards, 215f
Memory block, 70
Memory hierarchy, symmetric multiprocessor architecture, 204–209, 204f
data reuse and locality, 204–205
memory system performance, 207–209
Memory implementation, 474–475
Memory management, 350, 358–361
virtual address translation, 359–361
virtual memory, 359
virtual page addresses, 359
Memory system performance, symmetric multiprocessor architecture, 207–209
Memory technologies, HPC architecture, 58–62
early memory devices, 59–61
modern memory technologies, 61–62
Mesh decomposition, 333–334, 340f
Mesh tessellation, 364, 366f
Message-driven computation, 613–614
Message passing interface (MPI), 253–255
advance in, 606–607
example, 254–255
jobs, 185
message passing interface example, 254–255
MPI_Finalize, 254
mpi.h, 253
MPI_Init, 253
scripts, SLURM job scripting, 167
standards, 251–253
Metrics, 4–5
Miniapplications as benchmarks, 135–138, 137t
Mirroring (RAID 1), 535
Modern graphics processing unit architecture, 471–477, 472t
compute architecture, 471–474, 474f
interconnects, 475–476
memory implementation, 474–475
programming environment, 476–477

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 662

Monitoring hardware events, 398–407
perf, 398–404
performance application programming interface, 404–407
Moore, Gordon, 17
Moore's law, 37–38
Motivation and justification, for clusters, 84–85
Motorola MC68881, 459–461, 459f
MPI_Finalize, 254
mpi.h, 253
MPI_Init, 253
MPI+X, 607
Multicore petaflops, 37
Multigrid methods, 328–329
Multiprocessors, HPC architecture, 73–78
commodity clusters, 77–78
massively parallel processors, 76
shared-memory multiprocessors, 74–76, 74f
Multithreading, 613
symmetric multiprocessor architecture, 203

N

Nano, 646
NAS parallel benchmarks, 130–131, 130t
Navigating the file system, Linux, 641–645
Neodigital age, 37–38, 616–620
cellular automata, 618–619
dataflow, 617–618
neuromorphic, 619
quantum computing, 619–620
Nested parallel threads, 229f
Network attached storage, 543–544
Network file system, 562–565, 563f
Network interface controllers, 213–215
Ethernet, 213–214
InfiniBand, 215
Neuromorphic computing, 619
Node, commodity clusters, 93–94
Nonblocking point-to-point communication, 279–281
Nonuniform memory access (NUMA), 75–76
architectures, 75f
Numerical integration, 621–624

O

Offset update and query, 560
OpenACC, 483–487, 488f
C++ AMP, 486
compute unified device architecture, 485
directives, 492–505
atomics, 504–505
data management, 497–501
Kernels construct, 495–496
loop scheduling, 501–504
parallel construct, 493–495
variable scope, 504
environment variables, 491–492
library calls, 489–491
programming concepts, 487–489
OpenCL, 485–486
OpenMP
advances in, 607
jobs, 184–185
programming model, 226–231
runtime library and environment variables, 228–231
directives, 230–231
environment variables, 229–230
runtime library routines, 230
thread parallelism, 226–228
thread variables, 228
scripts, SLURM job scripting, 167
Operating systems, 347–349, 348f
commodity clusters, 98–99
memory management, 358–361
virtual address translation, 359–361
virtual memory, 359
virtual page addresses, 359
outcomes, 361
process management, 351–357
process control block, 353–354
process management activities, 354–355
process states, 352–353
scheduling, 355–357, 356f
structures and services, 349–351
file management, 350
I/O system management, 351
memory management, 350
process management, 349–350
secondary storage management, 351
system components, 349
threads, 357, 358f
Optical storage, 529–532, 530f

P

Package configuration and building, commodity clusters, 110–111
Parallel algorithms
divide and conquer, 287–291, 288f–291f
embarrassingly parallel, 292–293, 293f
fork–join, 286–287, 287f–288f
generic classes of, 286t
halo exchange, 294–299, 294f
advection equation using finite difference, 295–297, 296f–298f

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 663

Parallel algorithms (*Continued*)
sparse matrix vector multiplication, 297–299, 299f–300f
manager–worker, 291–292, 292f
outcomes, 310–311
overview, 285–286
permutation: Cannon's algorithm, 301–303, 301f–307f
task dataflow: breadth first search, 306–310, 307f–309f
Parallel architecture families, 48–50
Parallel construct, 493–495
Parallel "for," OpenMP programming model, 233–238
Parallel input/output, 330–332, 335f
Parallel programming modalities, commodity clusters, 97–98
Parallel threads/loops, OpenMP programming model, 231–240
parallel "for", 233–238
parallel threads, 231–232
private, 232–233
sections, 239–240
Parallelism, HPC architecture, 45–46
Parallelization, 334
ParaView, 379, 379f
Partial differential equations, 329
Pascal GP100 architecture, 473f
Pascaline mechanical calculator, 53f
Path expansion, Linux, 647–648
pbsnodes, 183
PCI bus, 209–213, 210f
connectors supporting, 211f
PCI express (PCIe), 210–212, 211f
slots, 212f
Peak performance, 17–18
perf, 398–404
Performance, 4–5, 16
degradation, 19–20
improvement, 20–21
monitoring, 341–342, 383–385
integrated performance monitoring toolkits, 407–410, 410f
monitoring hardware events, 398–407
perf, 398–404
performance application programming interface, 404–407
outcomes, 417–418
performance profiling, 390–398
essential gperftools, 391–398
significance of application profiling, 390–391
profiling in distributed environments, 411–417, 414f–415f, 417f
time measurement, 385–390
Performance application programming interface, 404–407
Performance profiling, 390–398
commodity clusters, 101
essential gperftools, 391–398
significance of application profiling, 390–391
Permutation: Cannon's algorithm, parallel algorithms, 301–303, 301f–307f
Personalized medicine and drug discovery, 11
Petaflops
applications, 328
multicore, 37
Pipeline parallelism, HPC architecture, 65–68, 67f
Pipelines, Linux, 649–650
Point-to-point messages, 258–262
example, 260–262
message-passing interface data types, 259, 260t
MPI Recv, 259–260
MPI send, 259
Portable batch system (PBS)
architecture, 173–174, 173f
cheat sheet, 186–187
commands, 174–183
pbsnodes, 183
qdel, 180
qstat, 180–182
job status query, 180–181
queue status query, 181
server status query, 182
qsub, 174–179
tracejob, 182–183
job scripting, 184–186
environment variables of interest, 185–186
MPI jobs, 185
OpenMP jobs, 184–185
overview, 172–173
Portable extensible toolkit for scientific computation, 327–328, 328t
Power, HPC architecture, 46–47
Predicting natural disasters, 12–14
Preemption, SLURM scheduling, 149–150
Private, OpenMP programming model, 232–233
Process control block, 353–354
Process management, 349–357
activities, 354–355
Linux, 661–662
process control block, 353–354
process management activities, 354–355
process states, 352–353
scheduling, 355–357, 356f
Process states, 352–353
Processor core architecture, symmetric multiprocessor architecture, 199–203, 200t
branch prediction, 201–202
execution pipeline, 200–201

forwarding, 202
instruction-level parallelism, 201
multithreading, 203
reservation stations, 202
Profiling in distributed environments, 411–417, 414f–415f, 417f
Programmability, HPC architecture, 48
Programming
concepts, 487–489
environment, 476–477
interfaces, commodity clusters, 97–98
high performance computing programming languages, 97
parallel programming modalities, 97–98

Q

qdel, 180
qstat, 180–182
job status query, 180–181
queue status query, 181
server status query, 182
qsub, 174–179
Quantum computing, 619–620

R

Redirection, Linux, 649–650
Reduction
OpenMP programming model, 244
operations, 274–276, 274t
Redundant array of independent disks, 532f, 534–541
hybrid redundant array of independent disks variants, 539–541
RAID 0: striping, 534–535
RAID 1: mirroring, 535
RAID 2: bit-level striping with hamming code, 536
RAID 3: byte-level striping with dedicated parity, 536–537
RAID 4: block-level striping with dedicated parity, 537
RAID 5: block-level striping with single distributed parity, 538
RAID 6: block-level striping with dual distributed parity, 539
Reliability, HPC architecture, 47–48
Remote access, Linux, 639–640
Reservation stations, symmetric multiprocessor architecture, 202
Resource management, commodity clusters, 99–100
debugger, 101
performance profiling, 101
visualization, 101–104
Running applications, commodity clusters, 113
Runtime library routines, OpenMP programming model, 230
Runtime system software, 615

S

sacct, 164–165
salloc, 160–161
sbatch, 161–162
Scalable library for eigenvalue problem computations, 328
Scalable linear algebra, 326
Scalable linear solvers, 328–329
Scaling, 18–19
scancel, 163–164
Scatter, 269–271
Scheduling, 355–357, 356f
Science, supercomputing impact on, 10–14
Scott, Steven, 35
Script components, SLURM job scripting, 166–167
Secondary storage
commodity clusters, 95
management, 351
Sections, OpenMP programming model, 239–240
Security, supercomputing impact on, 10–14
Sequencer controller, 70
Sequential data access, 555–556, 560
Serial advanced technology attachment (SATA), 215–218
connectors, 216f
interface variants, 218f
Serial quicksort algorithm, 289f
Shannon, Claude Elwood, 23
Shared neighbors, 581–582
Shared-memory multiprocessors, 227f
HPC architecture, 74–76, 74f
Signal processing, 334–341
SIMD array class of parallel computer architecture, 69f
Simple Linux Utility for Resource Management (SLURM)
cheat sheet, 171–172
job scripting, 166–171
concurrent applications, 167–169
environment variables, 169–171
MPI scripts, 167
OpenMP scripts, 167
script components, 166–167
scheduling, 149–150
elastic computing, 150
gang scheduling, 149
generic resources, 150
high-throughput computing, 150
preemption, 149–150
trackable resources, 150
sinfo, 165–166
Single directive, OpenMP programming model, 243
Single-bit full adder, 66f
Single-instruction
multiple data architecture, 69–70
multiple data array, 33, 69–73

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 665

Single-instruction (*Continued*)
Amdahl's law, 70–73
single-instruction, multiple data architecture, 69–70
Society, supercomputing impact on, 10–14
Software environment, commodity clusters, 98–104
operating systems, 98–99
resource management, 99–100
debugger, 101
performance profiling, 101
visualization, 101–104
Solid-state drive storage, 520–524, 521f–522f, 525t, 529t
SoLvers for Petaflop-applications, 328
Sparse matrix vector multiplication, 297–299, 299f–300f
Special character handling, Linux, 649
Speed, HPC architecture, 45
squeue, 162–163
srun, 151–160
Standard HPC community benchmarks, 120
Static random access memory (SRAM), 45, 61
Storage area networks, 541–542
Storage device technology, 514–532
hard-disk drives, 514–520, 515f, 517f, 519t
magnetic tape, 524–528, 527f–528f, 529t
optical storage, 529–532, 530f
solid-state drive storage, 520–524, 521f–522f, 525t, 529t
Streamlines, 364, 364f
Streams and file descriptors, conversion between, 561–562
Striping (RAID 0), 534–535
Structures and services, 349–351
file management, 350
I/O system management, 351
memory management, 350
process management, 349–350
secondary storage management, 351
system components, 349
Supercomputing, impact on science, society, and security, 10–14
accelerating innovation in manufacturing, 10–11
catalyzing fraud detection and market data analytics, 10
discovering oil and gas, 10
distributing oil and gas, 10
managing oil and gas, 10
personalized medicine and drug discovery, 11
predicting natural disasters, 12–14
understanding climate change, 12–14
Supercomputing problems, 7–8, 8t
Supernodal LU, 326–327
Sustained performance, 18
Symmetric memory processor (SMP), 55, 75, 194f
examples and characteristics, 192t
Symmetric multiprocessor architecture
Amdahl's law plus, 196–199, 197f–199f
architecture overview, 192–196
external I/O interfaces, 213–222
JTAG, 218–219
network interface controllers, 213–215
Ethernet, 213–214
InfiniBand, 215
serial advanced technology attachment, 215–218
universal serial bus, 220–222
memory hierarchy, 204–209, 204f
data reuse and locality, 204–205
memory hierarchy, 205–207, 206f
memory system performance, 207–209
outcomes, 222–223
overview, 191–192
PCI bus, 209–213, 210f
processor core architecture, 199–203, 200t
branch prediction, 201–202
execution pipeline, 200–201
forwarding, 202
instruction-level parallelism, 201
multithreading, 203
reservation stations, 202
Synchronization, OpenMP programming model, 241–243
barrier directive, 243
critical synchronization directive, 242
master directive, 242–243
single directive, 243
Synchronization collectives, 262–265
barrier synchronization, 263, 263f
example, 264–265
overview of collective calls, 262–263
Synchronization with storage device, 557
System area networks, commodity clusters, 94–95
System calls for file access, 554–558
data access with explicit offset, 556
file length adjustment, 556–557
file offset manipulation, 556
file open and close, 554–555
file status query, 557–558
sequential data access, 555–556
synchronization with storage device, 557
System components, 349
System monitors to aid debugging, 441–445
System-level checkpointing, 592–597

T

Tabulator built for the 1890 census, 54f
Task dataflow, parallel algorithms, 306–310, 307f–309f
Technology epochs, HPC architecture, 51–55
Tertiary storage, 544, 544f, 545t
Text tools, Linux, 659–661
Thermal control, 47

Thinking Machine CM-2, 34f
Thread parallelism, OpenMP programming model, 226–228
Thread variables, OpenMP programming model, 228
Threads, 357, 358f
Tic-Tac-Toe game, 631–635
Time measurement, 385–390
Timing in digital logic, 57f
Titan petaflops machine, 2f
Top 500 list, clusters impact on, 86–87
tracejob, 182–183
Trackable resources, SLURM scheduling, 150
Translation lookaside buffer (TLB), 360, 360f
Turing, Alan M., 23

U

Universal serial bus (USB), 220–222
  connector types, 221f
  devices connected in multitiered topology, 221f
Unprotected shared variable assessment, 433–434, 436f
User space and directory system, commodity clusters, 105–110
User-defined data types, 281–283

V

Vacuum tubes, von Neumann architecture in, 24–29
Valgrind, 430–431, 433t
Variable scope, 504
Variables, Linux, 650–653
Vector integration, 30–33
Vector processing, 30–33
  HPC architecture, 68, 68f
Vi, 645
Virtual address translation, 359–361
Virtual memory, 359
Virtual page addresses, 359
VisIt, 380, 380f
Visualization, 334, 363
  commodity clusters, 101–104
  foundational visualization concepts, 364
  Gnuplot, 365–368, 366f–370f
  Matplotlib, 369–372, 371f–372f
  outcomes, 381
  ParaView, 379, 379f
  VisIt, 380, 380f
  Visualization Toolkit, 372–379, 373f–374f, 377f
Visualization Toolkit, 372–379, 373f–374f, 377f
Volume rendering, 364, 365f
von Neumann, John, 28
  architecture in vacuum tubes, 24–29
  sequential processors, 62–64, 63f

W

Watanabe, Tadashi, 30–33
Weitek WTL 4167 FPU, 462f
Whirlwind vacuum-tube-based digital electronic computer, 59b
Wilkes, Maurice, 26b
Word count, 581
Workload organization, essential SLURM, 148, 148f
World's fastest machine, 608

HIGH PERFORMANCE COMPUTING

MODERN SYSTEMS AND PRACTICES

THOMAS STERLING, MATTHEW ANDERSON, MACIEJ BRODOWICZ

FOREWORD BY C. GORDON BELL

High Performance Computing: Modern Systems and Practices is a fully comprehensive and easily accessible treatment of high performance computing, covering fundamental concepts and essential knowledge while providing key skills training. Because an understanding of HPC is central to achieving advances in science, engineering, commerce, industry, national security, and the Internet itself, system engineers need guidance on how to properly leverage these vital resources. With this book, domain scientists learn how to use supercomputers as a key tool in their quest for new knowledge. Practicing engineers will discover how supercomputers can employ HPC systems and methods to the design and simulation of innovative products. Students can begin their careers with an understanding of possible directions for future research and development in HPC. Those who maintain and administer commodity clusters will find this textbook provides essential coverage of what HPC systems do and how they are used.

FEATURES

- Covers enabling technologies, system architectures and operating systems, parallel programming languages and algorithms, scientific visualization, correctness and performance debugging tools and methods, GPU accelerators and big data problems
- Provides numerous examples that explore the basics of supercomputing, while also providing practical training in the real use of high-end computers
- Helps users with informative and practical examples that build knowledge and skills through incremental steps
- Features sidebars of background and context to present a live history and culture of this unique field
- Includes online resources, such as recorded lectures from the authors' HPC courses

ABOUT THE AUTHORS

Thomas Sterling is Professor of Intelligent Systems Engineering at Indiana University. He serves as the Director of the Center for Research in Extreme Scale Technologies (CREST). He is most widely known for his pioneering work in commodity cluster computing as leader of the Beowulf Project for which he and colleagues were awarded the Gordon Bell Prize.

Matthew Anderson is Assistant Scientist at the Center for Research in Extreme Scale Technologies (CREST) at Indiana University.

Maciej Brodowicz is Assistant Scientist at the Center for Research in Extreme Scale Technologies (CREST) at Indiana University.

High Performance Computing
Parallel Programming

MK
MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
elsevier.com/books-and-journals

ISBN 978-0-12-420158-3

PATENT OWNER DIRECTSTREAM, LLC
EX. 2136, p. 668