Homayoun

Reference 24

HIGH PERFORMANCE COMPUTING

MODERN SYSTEMS AND PRACTICES

THOMAS STERLING, MATTHEW ANDERSON,
MACIEJ BRODOWICZ

MORGAN KAUFMANN

FOREWORD BY C. GORDON BELL

High Performance Computing

High Performance Computing
Modern Systems and Practices

Thomas Sterling
Matthew Anderson
Maciej Brodowicz
School of Informatics, Computing, and Engineering
Indiana University, Bloomington

Foreword by C. Gordon Bell

ELSEVIER

MK
MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-420158-3

Publisher: Katey Birtcher
Acquisition Editor: Steve Merken
Developmental Editor: Nate McFadden
Production Project Manager: Punithavathy Govindaradjane
Designer: Mark Rogers

Typeset by TNQ Books and Journals

Dedicated to

Dr. Paul C. Messina

Leader, colleague, collaborator, mentor, friend

Contents

Foreword xix
Preface xxi
Acknowledgments xxvii

CHAPTER 1 Introduction 1

1.1 High Performance Computing Disciplines 3
1.1.1 Definition 3
1.1.2 Application Programs 4
1.1.3 Performance and Metrics 4
1.1.4 High Performance Computing Systems 5
1.1.5 Supercomputing Problems 7
1.1.6 Application Programming 8
1.2 Impact of Supercomputing on Science, Society, and Security 10
1.2.1 Catalyzing Fraud Detection and Market Data Analytics 10
1.2.2 Discovering, Managing, and Distributing Oil and Gas 10
1.2.3 Accelerating Innovation in Manufacturing 10
1.2.4 Personalized Medicine and Drug Discovery 11
1.2.5 Predicting Natural Disasters and Understanding Climate Change 12
1.3 Anatomy of a Supercomputer 14
1.4 Computer Performance 16
1.4.1 Performance 16
1.4.2 Peak Performance 17
1.4.3 Sustained Performance 18
1.4.4 Scaling 18
1.4.5 Performance Degradation 19
1.4.6 Performance Improvement 20
1.5 A Brief History of Supercomputing 21
1.5.1 Epoch I—Automated Calculators Through Mechanical Technologies 22
1.5.2 Epoch II—von Neumann Architecture in Vacuum Tubes 24
1.5.3 Epoch III—Instruction-Level Parallelism 29
1.5.4 Epoch IV—Vector Processing and Integration 30
1.5.5 Epoch V—Single-Instruction Multiple Data Array 33
1.5.6 Epoch VI—Communicating Sequential Processors and Very Large Scale Integration 34
1.5.7 Epoch VII—Multicore Petaflops 37
1.5.8 Neodigital Age and Beyond Moore's Law 37
1.6 This Textbook as a Guide and Tool for the Student 38

1.7 Summary and Outcomes of Chapter 1 39
1.8 Questions and Problems 40
References 41

CHAPTER 2 HPC Architecture 1: Systems and Technologies 43
2.1 Introduction 44
2.2 Key Properties of HPC Architecture 44
2.2.1 Speed 45
2.2.2 Parallelism 45
2.2.3 Efficiency 46
2.2.4 Power 46
2.2.5 Reliability 47
2.2.6 Programmability 48
2.3 Parallel Architecture Families—Flynn's Taxonomy 48
2.4 Enabling Technology 51
2.4.1 Technology Epochs 51
2.4.2 Roles of Technologies 55
2.4.3 Digital Logic 55
2.4.4 Memory Technologies 58
2.5 von Neumann Sequential Processors 62
2.6 Vector and Pipelining 64
2.6.1 Pipeline Parallelism 65
2.6.2 Vector Processing 68
2.7 Single-Instruction, Multiple Data Array 69
2.7.1 Single-Instruction, Multiple Data Architecture 69
2.7.2 Amdahl's Law 70
2.8 Multiprocessors 73
2.8.1 Shared-Memory Multiprocessors 74
2.8.2 Massively Parallel Processors 76
2.8.3 Commodity Clusters 77
2.9 Heterogeneous Computer Structures 78
2.10 Summary and Outcomes of Chapter 2 78
2.11 Questions and Problems 80
References 82

CHAPTER 3 Commodity Clusters 83
3.1 Introduction 84
3.1.1 Definition of "Commodity Cluster" 84
3.1.2 Motivation and Justification for Clusters 84
3.1.3 Cluster Elements 85

3.1.4 Impact on Top 500 List ... 86
3.1.5 Brief History ... 88
3.1.6 Chapter Guide ... 90
3.2 Beowulf Cluster Project ... 91
3.3 Hardware Architecture ... 93
3.3.1 The Node ... 93
3.3.2 System Area Networks ... 94
3.3.3 Secondary Storage ... 95
3.3.4 Commercial Systems Summary ... 95
3.4 Programming Interfaces ... 97
3.4.1 High Performance Computing Programming Languages ... 97
3.4.2 Parallel Programming Modalities ... 97
3.5 Software Environment ... 98
3.5.1 Operating Systems ... 98
3.5.2 Resource Management ... 99
3.5.3 Debugger ... 101
3.5.4 Performance Profiling ... 101
3.5.5 Visualization ... 101
3.6 Basic Methods of Use ... 104
3.6.1 Logging On ... 104
3.6.2 User Space and Directory System ... 105
3.6.3 Package Configuration and Building ... 110
3.6.4 Compilers and Compiling ... 112
3.6.5 Running Applications ... 113
3.7 Summary and Outcomes of Chapter 3 ... 113
3.8 Questions and Exercises ... 114
References ... 114

CHAPTER 4 Benchmarking ... 115
4.1 Introduction ... 115
4.2 Key Properties of an HPC Benchmark ... 117
4.3 Standard HPC Community Benchmarks ... 120
4.4 Highly Parallel Computing Linpack ... 120
4.5 HPC Challenge Benchmark Suite ... 123
4.6 High Performance Conjugate Gradients ... 126
4.7 NAS Parallel Benchmarks ... 130
4.8 Graph500 ... 132
4.9 Miniapplications as Benchmarks ... 135
4.10 Summary and Outcomes of Chapter 4 ... 138
4.11 Exercises ... 139
References ... 139

CHAPTER 5 The Essential Resource Management **141**
5.1 Managing Resources 142
5.2 The Essential SLURM 146
5.2.1 Architecture Overview 147
5.2.2 Workload Organization 148
5.2.3 SLURM Scheduling 149
5.2.4 Summary of Commands 151
5.2.5 SLURM Job Scripting 166
5.2.6 SLURM Cheat Sheet 171
5.3 The Essential Portable Batch System 172
5.3.1 Portable Batch System Overview 172
5.3.2 Portable Batch System Architecture 173
5.3.3 Summary of PBS Commands 174
5.3.4 PBS Job Scripting 184
5.3.5 PBS Cheat Sheet 186
5.4 Summary and Outcomes of Chapter 5 187
5.5 Questions and Problems 189
References 190

CHAPTER 6 Symmetric Multiprocessor Architecture **191**
6.1 Introduction 191
6.2 Architecture Overview 192
6.3 Amdahl's Law Plus 196
6.4 Processor Core Architecture 199
6.4.1 Execution Pipeline 200
6.4.2 Instruction-Level Parallelism 201
6.4.3 Branch Prediction 201
6.4.4 Forwarding 202
6.4.5 Reservation Stations 202
6.4.6 Multithreading 203
6.5 Memory Hierarchy 204
6.5.1 Data Reuse and Locality 204
6.5.2 Memory Hierarchy 205
6.5.3 Memory System Performance 207
6.6 PCI Bus 209
6.7 External I/O Interfaces 213
6.7.1 Network Interface Controllers 213
6.7.2 Serial Advanced Technology Attachment 215
6.7.3 JTAG 218
6.7.4 Universal Serial Bus 220
6.8 Summary and Outcomes of Chapter 6 222
6.9 Questions and Exercises 223
References 224

CHAPTER 7 The Essential OpenMP **225**
7.1 Introduction 225
7.2 Overview of OpenMP Programming Model 226
7.2.1 Thread Parallelism 226
7.2.2 Thread Variables 228
7.2.3 Runtime Library and Environment Variables 228
7.3 Parallel Threads and Loops 231
7.3.1 Parallel Threads 231
7.3.2 Private 232
7.3.3 Parallel "For" 233
7.3.4 Sections 239
7.4 Synchronization 241
7.4.1 Critical Synchronization Directive 242
7.4.2 The Master Directive 242
7.4.3 The Barrier Directive 243
7.4.4 The Single Directive 243
7.5 Reduction 244
7.6 Summary and Outcomes of Chapter 7 245
7.7 Questions and Problems 246
Reference 247

CHAPTER 8 The Essential MPI **249**
8.1 Introduction 250
8.2 Message-Passing Interface Standards 251
8.3 Message-Passing Interface Basics 253
8.3.1 mpi.h 253
8.3.2 MPI_Init 253
8.3.3 MPI_Finalize 254
8.3.4 Message-Passing Interface Example—Hello World 254
8.4 Communicators 255
8.4.1 Size 256
8.4.2 Rank 256
8.4.3 Example 257
8.5 Point-to-Point Messages 258
8.5.1 MPI Send 259
8.5.2 Message-Passing Interface Data Types 259
8.5.3 MPI Recv 259
8.5.4 Example 260
8.6 Synchronization Collectives 262
8.6.1 Overview of Collective Calls 262
8.6.2 Barrier Synchronization 263
8.6.3 Example 264

8.7 Communication Collectives 265
8.7.1 Collective Data Movement 265
8.7.2 Broadcast 268
8.7.3 Scatter 269
8.7.4 Gather 271
8.7.5 Allgather 272
8.7.6 Reduction Operations 274
8.7.7 Alltoall 277
8.8 Nonblocking Point-to-Point Communication 279
8.9 User-Defined Data Types 281
8.10 Summary and Outcomes of Chapter 8 283
8.11 Exercises 283
References 284
CHAPTER 9 Parallel Algorithms **285**
9.1 Introduction 285
9.2 Fork–Join 286
9.3 Divide and Conquer 287
9.4 Manager–Worker 291
9.5 Embarrassingly Parallel 292
9.6 Halo Exchange 294
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
CHAPTER 10 Libraries **313**
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

CHAPTER 11 Operating Systems ... **347**
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

CHAPTER 12 Visualization ... **363**
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

CHAPTER 13 Performance Monitoring **383**
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

CHAPTER 14 Debugging **421**
14.1 Introduction 421
14.2 Tools 423
14.2.1 The GNU Debugger 423
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

CHAPTER 15 Accelerator Architecture **451**
15.1 Introduction 451
15.2 A Historic Perspective 454
15.2.1 Coprocessors 456
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

high performance computing to mission-critical problem domains of the US Department of Energy (DOE). He then founded and directed the Caltech Concurrent Supercomputing Facilities that staged the Intel Touchstone Delta massively parallel processor, the fastest computer in the world in 1991 and the prototype of a family of massively parallel processors that determined the future direction of high performance computing for the next 30 years. Caltech Concurrent Supercomputing Facilities evolved into the Caltech Center for Advanced Computing Research, at which two of the authors spent some of their most formative years. Paul was particularly instrumental and served as co-principal investigator for the pioneering NSF Teragrid and the National Virtual Observatory. He directed the DOE ASCI program for almost 3 years, building up the nation's high performance computing capabilities toward the leadership-scale computing it currently demonstrates. Most recently Paul Messina led the Exascale Computing Project, America's biggest undertaking in achieving exascale computing performance by the beginning of the 2020s. For some Paul has had a direct and meaningful effect on their individual careers. To author Thomas Sterling Paul has been a colleague, leader, mentor, and friend for many years.

Acknowledgments

This textbook would not have been possible in either form or quality without the many contributions, both direct and indirect, of a large number of friends and colleagues. It is derivative of first-year graduate courses taught at both Louisiana State University (LSU) and Indiana University (IU). A number of people contributed to these courses, including Chirag Dekate, Daniel Kogler, and Timur Gilmanov. Amy Apon, a professor at the University of Arkansas, partnered with LSU and taught this course in real time over the internet and helped to develop pedagogical material, including many of the exercises used. Now at Clemson University, she continued this important contribution using her technical and pedagogical expertise. Andrew Lumsdaine, then a professor at IU, cotaught the first version of this course at IU. Amanda Upshaw was instrumental in the coordination of the process that resulted in the final draft of the book, and directly developed many of the illustrations, graphics, and tables. She was also responsible for the glossary of terms and acronyms. Her efforts are responsible in part for the quality of this textbook.

A number of friends and colleagues provided guidance as the authors crafted early drafts of the book. These contributions were of tremendous value, and helped improve the quality of content and form to be useful for readers and students. David Keyes of KAUST reviewed and advised on Chapter 9 on parallel algorithms. Jack Dongarra provided important feedback on Chapter 4 on benchmarking.

This textbook reflects decades of effort, research, development, and experience by uncounted number of contributors to the field of high performance computing. While not directly involved with the creation of this text, many colleagues have contributed to the concepts, components, tools, methods, and common practices associated with the broad context of high performance computing and its value. Among these are Bill Gropp, Bill Kramer, Don Becker, Richard and Sarah Murphy, Jack Dongarra and his many collaborators, Satoshi Matsuoka, Guang Gao, Bill Harrod, Lucy Nowell, Kathy Yelick, John Shalf, John Salmon, and of course Gordon Bell. Thomas Sterling would like to acknowledge his thesis advisor (at MIT) Bert Halstead for his mentorship to become the contributor that he has become. Thomas Sterling also acknowledges Jorge Ucan, Amanda Upshaw, co-authors who made this book possible, and especially Paul Messina who is his colleague, role model, mentor, and friend without whom this book would never occurred. Matthew Anderson would like to thank Dayana Marvez, Oliver Anderson, and Beltran Anderson. Maciej Brodowicz would like to thank his wife Yuko Prince Brodowicz. The authors would like to thank Nate McFadden of Morgan-Kaufmann who provided enormous effort, guidance, and patience that made this textbook possible.

DEDICATION TO PAUL MESSINA, WRITTEN BY THOMAS STERLING

The authors are pleased to dedicate this book to Dr Paul C. Messina, in acknowledgment of and gratitude for his exceptional contributions to and leadership in the field of high performance computing over a career of more than 4 decades. It is impossible to capture fully the importance of his impact, but many of the significant national programs have benefited from his guidance. Dr Messina has been a visionary, a strategist, and a leader of programs, projects, organizations, initiatives, and, perhaps most importantly, the careers of individual scientists who would come to deliver technical accomplishments and leadership of their own. Dr Messina was the founding director of the Mathematics and Computer Science Division at Argonne National Laboratory, a leading institution applying

- Future engineers and HPC system developers, and
- HPC system administrators and data center managers.

HOW TO USE THIS TEXTBOOK

The textbook is designed to serve multiple distinct approaches to learning and education, depending on the needs of the specific students.

- To achieve a full and in-depth understanding of HPC from an initial starting position, the book can be read from cover to cover. Its order of presentation of topical chapters is organized so that each builds on the material of the previous ones in the areas of concepts, knowledge, and skills. The examples embedded within the text are sufficient to represent the distinct points such that they are accessible to the reader.
- At the other extreme, the textbook can serve as a tutorial by reading the chapters and sections with titles/headings beginning with "The Essential...". These units are intended to develop the reader's skill-sets with minimal background and contextual information.
- Emphasis can be achieved by selecting the critical path chapters (or sections). Four models of parallel computing are represented: throughput, message passing, shared memory, and accelerated. But in some cases only one of these is required by the student or educator, thus a student may only need to be immersed in a subset of the chapters offered. For example, a course may use OpenMP or MPI but not both. For basic job-stream parallelism, both of these can be side-stepped and instead the student is focused on SLURM or PBS for throughput computing.

VII. WORKING WITH THE REAL SYSTEM (CHAPTERS 11 AND 17–23)

The HPC system does not operate in a vacuum, and is of little value if it is not connected with the outside world. Throughout the book the reader is exposed to necessary bits of the system environment, but these chapters give a focused and comprehensive description of the operating system and its interface to the outside world. In particular, mass storage is described at both hardware and software levels for persistent storage of large blocks of data through the file system. As an example of the use of the file system, the map-reduce algorithm, which is very popular for big-data problems, is described in detail. The file system is also used to improve reliability through a method of checkpoint-restart. This technique periodically stores a snapshot of the intermediate data state of an application on mass storage in case a fault occurs in the system. Should this happen, the application program can be restarted not from the beginning but rather at the last known good checkpoint, thus saving a lot of time to get to a solution.

VIII. NEXT STEPS

At this point the reader will have come to the end of his or her introduction to HPC. But where does the reader go from here? There is much more to the field of such systems and their use than could be incorporated in any single textbook, although a good job of it has been done here. It is useful for the student to have a clear picture of what is out in front and, depending on one's interests or goals, which areas to pursue next. This chapter maps out the space of HPC beyond that contained in this text, and highlights the different areas as they relate to distinct professional objectives. But there is another dimension to the next steps: where is the field of HPC going itself, for it is changing very rapidly? The chapter concludes with a high-level description of the challenges facing HPC and the opportunities driving it forward.

WHO CAN BENEFIT FROM THIS TEXTBOOK?

This book is constructed so that the widest readership with diverse backgrounds can with a high probability of success take on the subject matter. For this reason it has been crafted with the minimal prerequisites of a working knowledge of programming in the C language and a familiarity of working within the context of a Unix-like operating system. But it is understood that even these requirements may be too stringent for some. For this reason, the appendix of the book includes two tutorials. One, "The Essential C," provides sufficient descriptive details in tutorial form to use the C programming language. It is not a primer of computer programming, as it is expected that the student has experience writing programs with some other programming language like Python, Java, Fortran, or MATLAB, but this tutorial is sufficient to support your needs through all the examples and exercises. Second, "The Essential Linux for Users," gives all the user interface descriptions and techniques that are required to fulfill all the tasks employed in this textbook.

This textbook may serve a broad community of possible readers including (but not limited to):

- Research scientists
- Computational scientists in end science, engineering, and societal domains
- HPC research faculty

consistency is assumed by hardware cache coherence. This part of the book describes this parallel execution model, the characteristics of shared-memory multiprocessors, and the OpenMP parallel programming language.

IV. MESSAGE-PASSING COMPUTING (CHAPTER 8)

For truly scalable parallel computing that may employ a million cores or more on a single application, the distributed-memory architecture and communicating sequential processing execution model is the dominant approach. This part of the book builds on topics associated with the nodes used for SMPs and the cluster approach previously described for throughput computing by adding the semantics of message passing, collective operations, and global synchronization. It is in this section that message-passing interface (MPI) is taught, the single most widely employed programming interface for scalable science and engineering applications.

V. ACCELERATING GPU COMPUTING (CHAPTERS 15 AND 16)

For certain widely used dataflow patterns, higher-level structures of specialized cores can provide exceptional performance and energy efficiency. Such subsystems, classified in the most general sense as "accelerators," can speed up applications by many times, sometimes by over an order of magnitude. Also referred to as GPGPUs, these often take the form of attached array processors, but in some cases are being integrated within single-socket packages or even the same die. This part of the textbook describes GPU structures, available products, and programming, with an emphasis on one programming interface, OpenACC.

VI. BUILDING SIGNIFICANT PROGRAMS (CHAPTERS 9, 10, AND 12–14)

By this point in the book the reader is well acquainted with the primary modes of HPC, knows the rules for the principal programming interfaces, and has hands-on experience with making basic parallel functions work within these frameworks. But for more complicated, more sophisticated, more useful, and frankly more professional supercomputing programs a number of additional methods and tools are required. This segment of the textbook takes the HPC novice from the beginner level to that of useful apprentice. Several key topics and skills are introduced here to give the student the necessary abilities to be useful in system design and application. First among these is a broad array of parallel algorithms for a diverse set of needs. Many of these are already made available in collections known as "libraries" that can save the application developer an enormous amount of time, if appropriately used. To get a program from its first draft to its final correct and efficient form requires a combined approach involving parallel debugging for correctness of answers and performance optimization through operation monitoring. Tools and methods for both are presented here, including the detailed skill-sets required. Finally, HPC runs tend to produce enormous amounts of data—as much as terabytes or petabytes of results in a single execution. Scientific visualization, the producing of images or even movies from such massive datasets, is the only practical way to achieve understanding of the results of a technical computing simulation. Examples of widely used tools for this purpose are presented, with essential techniques to make them useful.

of this distinction is that while much of the knowledge about this rapidly evolving field will change, and even become obsolete in some cases, the basic concepts offered are invariant with time and will serve the reader with strong long-term understanding even as the details of some specific machine or language may become largely irrelevant over time.

This textbook is organized first according to the four separate models of parallel computation, and then for each model according to the underlying concepts, the relevant knowledge with an emphasis on system architectures that support them, and the skills required to train the reader in how each class of system is programmed. In preparation for this approach, some initial material, including the introductory chapter, provides the basic premises and context upon which the textbook is established. Each of the four parallel computing models is described in terms of concepts, knowledge details, and programming skills. But while this covers a large part of the useful information needed to understand and program HPC systems, it misses some of the cross-cutting topics related to environments and tools that are an important, even pervasive, aspect of the full context of a system that makes it truly useful beyond the limits of an idealized beginner's viewpoint. After all, the intent of the textbook is to give the reader an effective working ability to take advantage of supercomputers in the professional workplace for diverse purposes. Thus a number of important and useful tools and methods of their use are given in an effective order. Finally, the reader is given a clear picture of the wide field of HPC, and where within this broader context the subject matter of this book fits. This can be used to guide planning for future pursuits and more advanced courses selected in part based on readers' ultimate professional goals. The overall structure and flow of this textbook are summarized below.

I. INTRODUCTORY AND BASIC IDEAS (CHAPTERS 1 AND 4)

These chapters provide a firm grounding on the basics, including an introduction to the domains of execution models, architecture concepts, performance and parallelism metrics, and the dominant class of parallel computing systems (commodity clusters). They give a first experience with running parallel programs through the use of a special kind of benchmarks that allow measurement and comparisons among different HPC systems. It is here that a sense of the history, the evolution of the contributing ideas, and the culture of the field is first given to the reader.

II. THROUGHPUT COMPUTING FOR JOB-STREAM PARALLELISM (CHAPTERS 5 AND 11)

Although among the simplest ways to take advantage of parallel computers, throughput computing (also referred to as capacity computing) as widely used is sufficient for many objectives and workflows. It can also prove to be among the most efficient, as it usually exhibits the most coarse-grained tasks and a minimum of control overheads. Widely used middleware that manages job-stream workloads such as SLURM and PBS are given in tutorial form for both independent jobs and related sets, such as parameter sweeps and Monte Carlo simulations.

III. SHARED-MEMORY MULTITHREADED COMPUTING (CHAPTERS 6 AND 7)

One of the dominant models of user parallel processing is task (or thread) parallelism in the context of shared memory. All the user data can be directly accessed by any of the user threads, and sequential

ORGANIZATION OF THIS BOOK

This textbook serves as a bridge between the reader's initial curiosity, interests, and requirements in HPC and the ultimate knowledge, capabilities, and proficiency to be acquired through its study. It is a starting point for those in pursuit of a number of different possible professional paths that share a common foundation in the nature and use of these state-of-the-art systems. Whether the reader intends ultimately to be able to build hardware or software systems, use such systems as a critical tool in the pursuit of other fields in science, engineering, commerce, or security, conduct research to devise future means of pushing the state of the art in HPC, or administer, manage, and maintain HPC systems for other users, the textbook is structured to create a seamless flow of topics, each benefiting from those preceding while contributing to the foundations supporting those following. Thus the book presents its major subjects in an order that provides early basic skills of HPC use even as it conveys underlying concepts upon which a deeper understanding of these complex systems and their use is based. Where necessary, an introductory view of a topic is given with enough information to consider other topics that are dependent, only to return in greater depth in later chapters. The readers' understanding and capabilities are ratcheted up through incremental enhancement across the diversity of interrelated topical areas.

The textbook is about computing performance. For current and next-generation systems, this means the use and exploitation of workload parallelism to achieve scalability and the means of managing data to achieve efficiency of operation. The four principal overarching subject domains are listed below.
- System hardware architecture, and enabling technologies.
- Programming models, interfaces, and methods.
- System software environments, support, and tools.
- Parallel algorithms and distributed data structures.

This would suggest an obvious pedagogical organization of the textbook based on a logical flow. But there is another dimension to HPC: alternative strategies for organizing and coordinating parallelism and data management, and the roles of each of the component layers that contribute to them. This book presents four major strategies.
- Job stream parallelism, throughput, or capacity computing.
- Communicating sequential processes, or message passing.
- Multiple-threaded shared memory.
- SIMD or graphics processing unit (GPU) accelerated.

From a pedagogical perspective, the authors wish to convey three kinds of information to facilitate the learning process and hopefully also the enjoyment of the reader. At the foundational level are the concepts that establish understanding of the underlying principles that guide the form and function of HPC. There is a lot of basic information as well as some cultural (who, what, when) facts making up the necessary collection of knowledge that provides the framework (scaffolding) of the field. Finally, there are the skill-sets that teach how to do things. While admittedly not orthogonal to each other, the textbook approaches the presentation of all the material in each case as one of these three forms. For example, chapters with headings that begin "The Essential..." (such as "The Essential OpenMP") are crafted as skills modules with a tutorial presentation style for easiest learning. While the mixing of concepts and knowledge is unavoidable, separate sections emphasize one or the other. The importance

Preface

THE PURPOSE OF THIS TEXTBOOK

High performance computing (HPC) is a multidisciplinary field combining hardware technologies and architecture, operating systems, programming tools, software, and end-user problems and algorithms. Acquiring the necessary concepts, knowledge, and skills for capable engagement within HPC routinely involves an apprenticeship at one of a few rarefied sites with the essential experts, facilities, and mission objectives. Whether one's goals are associated with specific end-user domains such as science, engineering, medicine, or commercial applications, or focused on the enabling systems' technologies and methodologies that make supercomputing effective, the entry-level practitioner must embrace a wide range of distinct but interrelated and interdependent areas that require an understanding of their synergies to yield the necessary expertise. The study material could easily encompass a dozen or more books and manuals, but even together they would not deliver the necessary perspective that fully embodies the field as a whole and guides the student in pursuit of an effective path to achieve sufficient expertise.

This textbook is designed to bridge the gap between myriad sources of narrow focus and the need for a single source that spans and interconnects the range of disciplines comprising the HPC field. It is an entry-level text requiring a minimum of prerequisites, but provides a full understanding of the domains and their mutual effects that make supercomputing an interdisciplinary field. From a practical point of view, this textbook builds valuable and specific skill-sets for parallel programming, debugging, performance monitoring, system resource usage and tools, and result visualization among other useful techniques. These skills are provided in the reinforcing context of basic foundational concepts of prolonged relevance, and knowledge of detailed attributes of hardware and software system components more likely to evolve over time.

The textbook is chartered as support for a single-semester course for beginners to prepare themselves for a diversity of roles in supercomputing to pursue their chosen professional career goals. It is appropriate for future computational scientists who are dedicated to the use of supercomputers to solve science, engineering, or societal-domain applications, among others. It provides a base-level description of possible target capabilities for system designers and engineers in hardware and software. It also is a foundation for those who wish to proceed as researchers in supercomputing itself, as an introductory presentation of conventional systems and practices as well as a representation of the challenges facing this exciting domain of exploration. The book is equally appropriate for those engaged in supporting supercomputing environments, such as data centers and system administrators, operators, and management. In informing future professionals, the textbook can be used in multiple ways. It serves as a reference work of basic information for supercomputing. It provides a sequence of lecture content for classroom delivery. It supports a hands-on approach with substantial examples, all of which can be executed on parallel computers, and exercises to guide students as they learn by doing. It makes clear where skill-sets and training are presented, with an easy-to-learn tutorial style. Concepts are presented in a detailed but accessible form to establish the "why" of methods conveyed and assist future users in decision-making based on fundamental truths, factors, and sensitivities. Finally, this book unifies within the same context the many sets of facts associated with the multiplicity of subdisciplines that in combination make up the field of supercomputing.

Fortunately, in the mid-1980s the "killer microprocessor" arrived, demonstrating cost effectiveness and unlimited scaling just by interconnecting increasingly powerful computers. Unfortunately, this multicomputer era has required abandoning both the single memory and the single sequential program ideal of Fortran. Thus "supercomputing" has evolved from a hardware engineering design challenge of the single (mono-memory) computer of the Seymour Cray era (1960—95) to a software engineering design challenge of creating a program to run effectively using multicomputers. Programs first operated on 64 processing elements (1983), then 1000 elements (1987), and now 10 million (2016) processing elements in thousands of fully distributed (mono-memory) computers in today's multicomputer era. So in effect, today's high performance computing (HPC) nodes are like the supercomputers of a decade ago, as processing elements have grown 36% per year from 1000 computers in 1987 to 10 million processing elements (contained in 100,000 computer nodes).

High Performance Computing is the essential guide and reference for mastering supercomputing, as the authors enumerate the complexity and subtleties of structuring for parallelism, creating, and running these large parallel and distributed programs. For example, the largest climate models simulate ocean, ice, atmosphere, and land concurrently created by a team of a dozen or more domain scientists, computational mathematicians, and computer scientists.

Program creation includes understanding the structure of the collection of processing resources and their interaction for different computers, from multiprocessors to multicomputers (Chapters 2 and 3), and the various overall strategies for parallelization (Chapter 9). Other topics include synchronization and message-passing communication among the parts of parallel programs (Chapters 7 and 8), additional libraries that form a program (Chapter 10), file systems (Chapter 18), long-term mass storage (Chapter 17), and components for the visualization of results (Chapter 12). Standard benchmarks for a system give an indication of how well your parallel program is likely to run (Chapter 4). Chapters 16 and 17 introduce and describe the techniques for controlling accelerators and special hardware cores, especially GPUs, attached to nodes to provide an extra two orders of magnitude more processing per node. These attachments are an alternative to the vector processing units of the Cray era, and typified by the Compute Unified Device Architecture, or CUDA, model and standard to encapsulate parallelism across different accelerators.

Unlike the creation, debugging, and execution of programs that run interactively on a personal computer, smartphone, or within a browser, supercomputer programs are submitted via batch processing control. Running a program requires specifying to the computer the resources and conditions for controlling your program with batch control languages and commands (Chapter 5), getting the program into a reliable and dependable state through debugging (Chapter 14), checkpointing, i.e., saving intermediate results on a timely basis as insurance for the computational investment (Chapter 20), and evolving and enhancing a program's efficacy through performance monitoring (Chapter 13).

Chapter 21 concludes with a forward look at the problems and alternatives for moving supercomputers and the ability to use them to petascale and beyond. In fact, the only part of HPC not described in this book is the incredible teamwork and evolution of team sizes for writing and managing HPC codes. However, the most critical aspect of teamwork resides with the competence of the individual members. This book is your guide.

Gordon Bell
October 2017

Foreword

High Performance Computing is a needed follow-on to Becker and Sterling's 1994 creation of the Beowulf clusters recipe to build scalable high performance computers (also known as a supercomputers) from commodity hardware. Beowulf enabled groups everywhere to build their own supercomputers. Now with hundreds of Beowulf clusters operating worldwide, this comprehensive text addresses the critical missing link of an academic course for training domain scientists and engineers—and especially computer scientists. Competence involves knowing exactly how to create and run (e.g., controlling, debugging, monitoring, visualizing, evolving) parallel programs on the congeries of computational elements (cores) that constitute today's supercomputers.

Mastery of these ever-increasing, scalable, parallel computing machines gives entry into a comparatively small but growing elite, and is the authors' goal for readers of the book. Lest the reader believes the name is unimportant: the first conference in 1988 was the ACM/IEEE Supercomputing Conference, also known as Supercomputing 88; in 2006 the name evolved to the International Conference on High Performance Computing, Networking, Storage, and Analysis, abbreviated SCXX. About 11,000 people attended SC16.

It is hard to describe a "supercomputer," but I know one when I see one. Personally, I never pass up a visit to a supercomputer having seen the first one in 1961—the UNIVAC LARC (Livermore Advanced Research Computer) at Lawrence Livermore National Laboratory, specified by Edward Teller to run hydrodynamic simulations for nuclear weapons design. LARC consisted of a few dozen cabinets of densely packed circuit board interconnected with a few thousand miles of wires and a few computational units operating at a 100 kHz rate. In 2016 the largest Sunway Light supercomputer in China operated a trillion times faster than LARC. It consists of over 10 million processing cores operating at a 1.5 GHz rate, and consumes 15 MW. The computer is housed in four rows of 40 cabinets, containing 256 processing nodes. A node has four interconnected 8 MB processors, controlling 64 processing elements or cores. Thus the 10.6 million processing elements deliver 125 peak petaflops, i.e., 160 cabinets $\times$ 256 physical nodes $\times$ 4 computers $\times$ (1 control + 8 $\times$ 8) processing elements or cores with a 1.31 **PB** memory (160 $\times$ 256 $\times$ 4 $\times$ 8 GB). Several of the Top 500 supercomputers have O(10,000) computing nodes that connect and control graphic processing units (GPUs) with O(100) cores. Today's challenge for computational program developers is designing the architecture and implementation of programs to utilize these megaprocessor computers.

From a user perspective, the "ideal high performance computer" has an *infinitely fast clock*, executes a *single instruction stream* program operating on data stored in an *infinitely large and fast single-memory*, and comes in any size to fit any budget or problem. In 1957 Backus established the von Neumann programming model with Fortran. The first or "Cray" era of supercomputing from the 1960s through the early 1990s saw the evolution of hardware to support this simple, easy-to-use ideal by increasing processor speed, pipelining an instruction stream, processing vectors with a single instruction, and finally adding processors for a program held in the single-memory computer. By the early 1990s evolution of a single computer toward the ideal had stopped: clock speeds reached a few GHz, and the number of processors accessing a single memory through interconnection was limited to a few dozen. Still, the limited-scale, multiple-processor shared memory is likely to be the most straightforward to program and use!

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

Appendix A: Essential C 623
Appendix B: Essential Linux 639
Glossary 667
Index 677

17.5 Summary and Outcomes of Chapter 17 ... 545
17.6 Questions and Problems ... 546
References ... 547

CHAPTER 18 File Systems ... **549**
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
18.2.2 Buffered File I/O ... 559
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

CHAPTER 19 MapReduce ... **579**
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

CHAPTER 20 Checkpointing ... **591**
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

CHAPTER 21 Next Steps and Beyond ... **605**
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

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

CHAPTER 16 The Essential OpenACC 483
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

CHAPTER 17 Mass Storage 509
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
17.4.2 Storage Area Networks 541
17.4.3 Network Attached Storage 543
17.4.4 Tertiary Storage 544

CHAPTER

INTRODUCTION

1

CHAPTER OUTLINE

1.1 High Performance Computing Disciplines 3
1.1.1 Definition 3
1.1.2 Application Programs 4
1.1.3 Performance and Metrics 4
1.1.4 High Performance Computing Systems 5
1.1.5 Supercomputing Problems 7
1.1.6 Application Programming 8
1.2 Impact of Supercomputing on Science, Society, and Security 10
1.2.1 Catalyzing Fraud Detection and Market Data Analytics 10
1.2.2 Discovering, Managing, and Distributing Oil and Gas 10
1.2.3 Accelerating Innovation in Manufacturing 10
1.2.4 Personalized Medicine and Drug Discovery 11
1.2.5 Predicting Natural Disasters and Understanding Climate Change 12
1.3 Anatomy of a Supercomputer 14
1.4 Computer Performance 16
1.4.1 Performance 16
1.4.2 Peak Performance 17
1.4.3 Sustained Performance 18
1.4.4 Scaling 18
1.4.5 Performance Degradation 19
1.4.6 Performance Improvement 20
1.5 A Brief History of Supercomputing 21
1.5.1 Epoch I—Automated Calculators Through Mechanical Technologies 22
1.5.2 Epoch II—von Neumann Architecture in Vacuum Tubes 24
1.5.3 Epoch III—Instruction-Level Parallelism 29
1.5.4 Epoch IV—Vector Processing and Integration 30
1.5.5 Epoch V—Single-Instruction Multiple Data Array 33
1.5.6 Epoch VI—Communicating Sequential Processors and Very Large Scale Integration 34
1.5.7 Epoch VII—Multicore Petaflops 37
1.5.8 Neodigital Age and Beyond Moore's Law 37

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00001-0

1.6 This Textbook as a Guide and Tool for the Student 38
1.7 Summary and Outcomes of Chapter 1 39
1.8 Questions and Problems 40
References 41

Supercomputing, which means supercomputers and their application, is among the most important developments of the modern age, with unequaled impact across a vast diversity of fields of inquiry and practical effect. From the extremes of arcane sciences to the most immediate practical concerns, supercomputers play an essential role in the progress and advancement of human capabilities, environments, and understanding. No other single technology in the history of humanity has experienced a similar rate of growth, even in its relatively short existence. Within the span of a single human lifetime, supercomputers have expanded their ability to perform calculations by a factor of 10 trillion or 13 orders of magnitude, and this is a conservative estimate. From less than a 1000 basic operations per second in the late 1940s to today's performance in excess of a 100 quadrillion floating-point operations per second (over 100 petaflops), supercomputer speed has steadily improved by about a factor of 200 times every decade through a series of advances in technology, architecture, programming methods, algorithms, and system software (Fig. 1.1). High performance computing (HPC), synonymous with supercomputing, is a principal means of exploration complementing empirical methods used for more than 2 millennia and theory practiced in the age of enlightenment of the last 4 centuries. As the "third pillar" of investigation, supercomputing enables new paths of inquiry, new techniques of design, and new methods of operating process. Even discoveries correctly credited to other classes of tools and instrumentation, such as giant telescopes or particle accelerators, require the use of supercomputers as well to produce their final results through data analysis (sometimes referred to as "big data"). It can be asserted that supercomputing allows us to understand the past, to control the present, and in limited cases to predict the future.

The skills required to employ HPC are multiple and complex, while the means of acquiring such skills to a sufficient degree require potentially years of study and experience at least in normal practice.

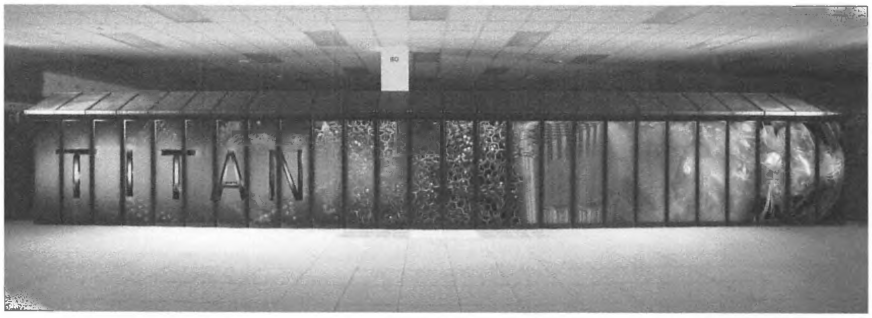


FIGURE 1.1

The Titan petaflops machine fully deployed at Oak Ridge National Laboratory in 2013. It takes up more than 4000 sq ft and consumes approximately 8 MW of electrical power. It has a theoretical peak performance of over 27 petaflops and delivers 17.6 petaflops R_{max} sustained performance for the highly parallel Linpack (HPL) benchmark. This architecture includes Nvidia graphics processing unit accelerators.

Photo courtesy of Oak Ridge National Laboratory, US Dept. of Energy

This often means lengthy apprenticeships in research facilities in academia, industry, or national laboratories. There are many books written to teach particular programming languages; others describe in detail the structures and instruction sets of computer architectures; and still others discuss system software such as operating systems. But missing has been a single textbook that serves as an entry-level presentation of all these elements and their interrelationships in one place, combined with guided hands-on experience. This work, *High Performance Computing*, is developed as a carefully crafted synthesis of relevant elements of related disciplines, all of which contribute in critical ways to supercomputing and its use. This book presents the foundation concepts, in-depth relevant knowledge, and detailed skills that together will give you a meaningful understanding of HPC and an initial set of techniques to make you an effective, albeit incipient, practitioner in its use. Throughout this text the best practices employed by the community are presented with training, so you learn to do, the *how*, even as you are gaining understanding of the *what* and the *why*.

This textbook provides a comprehensive introduction to the field of HPC. It is presented in a form that will be both intellectually rewarding and practical in teaching useful basic skills. It combines perspectives about supercomputing concepts, knowledge about supercomputers, and techniques for using and programming supercomputers. But teaching a complex subject like HPC is challenging, in that just about everything is defined in terms of and relates to everything else. Yet by the nature of pedagogy, material must be presented in some sequential order. This first chapter is a brief introductory presentation of the essential elements of HPC to provide an overview of everything; a first pass that will allow successive in-depth chapters to be related to this broad context.

The chapter looks at the many facets which comprise HPC. The importance of the material is that it provides a complete, albeit simplified, perspective of HPC so that more detailed discussion of specifics can be understood within the full context. Because no piece makes sense without the others, almost all areas are briefly introduced in this chapter. To reinforce the interrelated broad-brush presentation of issues, this chapter concludes with a history of the field and its rapid evolution.

1.1 HIGH PERFORMANCE COMPUTING DISCIPLINES

As previously noted, HPC is really a collection of multiple interrelated disciplines, each providing an important aspect of the total field. To master HPC as a useful tool is to develop an understanding and associated skills in each of these corresponding areas. These broad areas are described here, including a formal definition of "high performance computing" that applies throughout the treatment of the field, end-user application problems that are the intended purpose of HPC across a wide range of science, engineering, societal, and security domains, the core concept of performance which is the distinguishing characteristic of HPC compared to other forms of computing, the hardware and software components that make up an HPC system, environments, tools, application programming, and the interfaces used. Each of these is presented in some detail in the following sections and together form a major portion of the concepts, knowledge content, and skills comprising this textbook.

1.1.1 DEFINITION

HPC is a field of endeavor that relates to all facets of technology, methodology, and application associated with achieving the greatest computing capability possible at any point in time and technology. It engages a class of electronic digital machines referred to as "supercomputers" to perform a wide array of computational problems or “applications” (alternatively “workloads”) as fast as is possible. The action of performing an application on a supercomputer is widely termed “supercomputing” and is synonymous with HPC.

1.1.2 APPLICATION PROGRAMS

The purpose of HPC is to derive answers to questions that cannot be adequately addressed alone through means of empiricism, theory, or even widely available or accessible commercial computers (e.g., enterprise servers). Historically supercomputers have been applied to science and engineering, and the methodology has been described as the “third pillar of science” alongside and complementing both experimentation (empiricism) and mathematics (theory). But the range of problems that supercomputers can tackle extends far beyond classical scientific and engineering studies to include challenges in socioeconomics, big-data management and learning, process control, and national security. An application, then, is both the problem to be solved and the body of “code” or collection of ordered computing instructions that represent the means of solving the problem. The code is the means by which the user conveys to the supercomputer how it is to perform the necessary computations to achieve the objectives of the problem. The full set of code used is a “computer program” or just “program”, and the person developing the application code is the “programmer”.

1.1.3 PERFORMANCE AND METRICS

While the notion of performance may be intuitive, it is not simple. There is no single measure of performance that fully reflects all aspects of the quality of computer operation. A “metric” is a quantifiable observable operational parameter of a supercomputer. Multiple perspectives and related metrics are routinely applied to characterize the behavioral properties and capabilities of an HPC system. Two basic measures are employed individually or in combination and in differing contexts to formulate the values used to represent the quality of a supercomputer. These two fundamental measures are “time” and “number of operations” performed, both under prescribed conditions.

For HPC the most widely used metric is “floating-point operations per second” or “flops”. A floating-point operation is an addition or multiplication of two real (or floating-point) numbers represented in some machine-readable and manipulatable form. Because supercomputers are so “powerful”, to describe their capability would require phrases like “a trillion or quadrillion operations per second”. The field adopts the same system of notation as science and engineering, using the Greek prefixes kilo, mega, giga, tera, and peta to represent 1000, 1 million, 1 billion, 1 trillion, and 1 quadrillion, respectively. The first supercomputers barely achieved 1 kiloflops (Kflops). Today’s fastest supercomputer exhibits a peak performance in the order of 125 petaflops. The laptop computer upon which this textbook was written has a peak performance of a few gigaflops. A supercomputer is millions of times more powerful than a laptop by this metric.

The true capability of a supercomputer is its ability to perform real work, to achieve useful results toward an end goal such as simulating a particular physical phenomenon (e.g., colliding neutron stars to determine resulting electromagnetic burst signatures). A better measure than flops is how long a given problem takes to complete. But because there are literally thousands (millions?) of such problems, this measure is not particularly useful broadly. Thus the HPC community selects specific problems around which to standardize. Such standardized application programs are “benchmarks”.

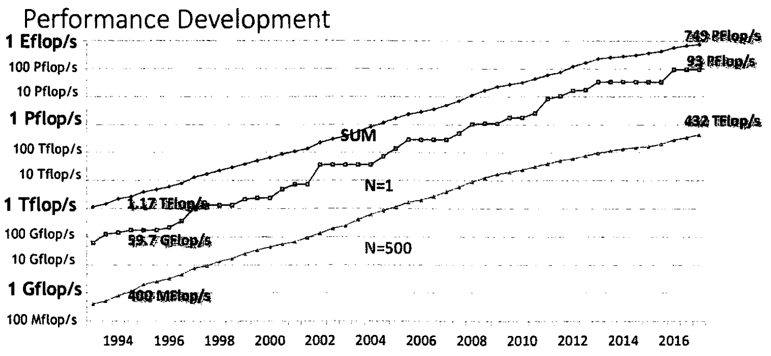


FIGURE 1.2

The evolution of the R_{max} from the HPL benchmark for supercomputing systems in the Top 500 list since the list began in 1993. The top line indicates the cumulative performance of all the computers in the list. The middle line shows the performance of the number one computer in the list. The bottom line shows the performance of the last computer in the list (number 500).

Image courtesy Erich Strohmaier

One particularly widely used supercomputer benchmark is "Linpack", or more precisely the "highly parallel Linpack" (HPL), which solves a set of linear equations in dense matrix form [1]. A benchmark gives a means of comparative evaluation between two independent systems by measuring their respective times to perform the same calculation. Thus a second way to measure performance is time to completion of a fixed problem. The HPC community has selected HPL as a means of ranking supercomputers, as represented by the "Top 500 list" begun in 1993 (Fig. 1.2). But other benchmarks are also employed to stress certain aspects of a supercomputer or represent a certain class of programs.

1.1.4 HIGH PERFORMANCE COMPUTING SYSTEMS

The most visible aspect of the field of HPC is the high performance computers, or simply supercomputers, themselves. Today these machines appear as rows upon rows of many racks taking up thousands of square feet and consuming potentially multiple megawatts of electrical power. To be in the presence of one (which often means to be literally inside it) offers a whole other experience in terms of noise, rapidly shifting temperature gradients, and many blinking lights. Even the most staid observer cannot help but be awestruck by the impressive massiveness of such systems, the engineering by which they are achieved, the commitment they represent to the edges of computing capability, and the problems only they can solve. And beyond what is visible even to the not-so-casual observer is the infrastructure that supports the operation of these systems, much of which is below floors, in adjacent rooms, and outside the building that houses the machine. The deployment of a state-of-the-art supercomputer is truly a major engineering undertaking involving time, expense, and expertise, as well as responsible management and maintenance throughout the lifetime of the system. And yet the visible, audible, and other sensory experiences barely reflect the true nature of the accomplishment embodied by these machines. At the heart of the HPC system is the structure and organization of its myriad components and the semantics or rules by which they operate and perform the user applications offered to them. Even more than the hardware, the HPC system is a vast array of software components that control the hierarchy of the physical components and manage the user workloads. If the physical hardware, racks and all, is what the visitor experiences, the system software, its interfaces, and functionality are what the user experiences (usually at a location far from the physical machine) when developing and running the applications and analyzing the results.

In one important sense, the high performance computer system has basic functionality and subsystems in common with the laptop personal computer upon which this textbook was written. These principal capabilities, shared by both extremes, include the following.

- The operational functions that transform input data values to output results.
- The internal memory that stores the data upon which the system operates.
- The communication channels through which intermediate data is transferred between different components and subsystems during application execution.
- The control hardware that coordinates the interoperability among the constituent components and subsystems.
- The mass storage that organizes and holds the persistent data, system software, and application programs.
- The input/output (I/O) channels and interfaces (like the keyboard I am typing on and the screen I am looking at) that connect users to the system.

Similarly, the software of an HPC system has much in common with the desk-side workstation or departmental enterprise server. Like these more pervasive albeit more modest computers, the supercomputer has a software structure that serves many of the same purposes of interface, control, and functionality, including but not limited to the following.

- The operating system that manages all aspects of the machine and its operation.
- The compilers that translate application programs written in human-readable syntactic languages (and other interfaces) to machine-readable binary code.
- File systems that present a logical abstraction of mass storage and organize the data on mass-storage devices (like hard-disk drives).
- The myriad software drivers of the I/O devices by which the computer communicates with the external world and users.
- The many tools that make up much of the expected user environments.

What distinguishes an HPC system from a conventional computer is the organization, interconnectivity, and scale of the component resources and the ability of the supporting software to manage the operation of the system at that scale (Fig. 1.3). By scale is meant the degree of physical and logical parallelism, i.e., the replication of key physical components such as processors and memory banks and the delineation of a number of tasks to be performed simultaneously. While even a single socket laptop incorporates some parallelism, an HPC system is structured in far more levels, each of which is usually much more substantial (but there are exceptions to this). It is this parallel organization, the methods by which the constituent subsystems are coordinated to solve a shared problem, and the additional functionality of the system software and programming models providing

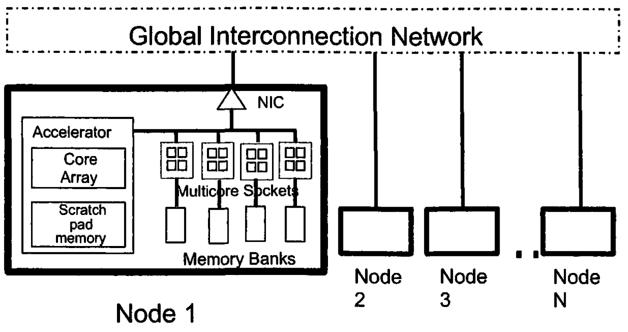


FIGURE 1.3

HPC systems are distinguished from a conventional computer by the organization, interconnectivity, and scale of the many component resources illustrated here. A "node" incorporates all the functional elements required for computation, and is highly replicated to achieve large scale.

such management that differentiate the supercomputer from its smaller counterparts. But from the viewpoint of the programmer, it is the need to think in parallel (many things happening at the same time) and distributed (things happening in different places separated by distance) that differentiates the supercomputer from the day-to-day computer [2]. This requires knowledge and skill in employing programming interfaces that expose and exploit application parallelism and algorithms that permit simultaneous operation of many parts of the computation contributing to the final answer.

1.1.5 SUPERCOMPUTING PROBLEMS

The field of supercomputing was born in the midst of revolutionary advances in experimental nuclear research, and has since grown to affect nearly all research fields driven by experiment. Because the genesis of supercomputing lies in simulating problems driven by nuclear physics, many supercomputing problems are framed in the context of tracking large systems of particles consisting of different species that may interact with one another and are not in equilibrium. Such nonequilibrium problems are generally difficult to compute analytically and can be very costly to explore experimentally. Consequently, these types of problems frequently appear on supercomputers, because of both the high-resolution probing ability of the simulation and the substantially reduced cost at which the computational experiment can be conducted [3].

Another class of supercomputing problem that overlaps with tracking large systems of particles with pair-wise interactions is the class that solves some set of partial-differential equations. For instance, a large fraction of supercomputing time is spent solving the Navier–Stokes equations for fluid flow because of their relevance to many engineering problems. As a second example, the direct detection of an astrophysical source of gravitational radiation in 2015 by the LIGO Scientific Collaboration was supported by millions of hours of supercomputing resources solving the Einstein field equations to simulate the merger of binary black holes.

Table 1.1 Supercomputing Problem Representatives and How They Are Used in Academia, Industry, and Government

Supercomputing Problem Representatives	Academia	Industry	Government
Solution of partial-differential equations	Navier–Stokes equations, Einstein equations, Maxwell equations	Black–Scholes equation, Navier–Stokes equations for compressible flow, oil reservoir modeling	Weather prediction, hurricane modeling, storm-surge modeling, sea-ice modeling
Large systems with pair-wise force interactions	Cosmology, molecular dynamics simulations	Medicine development, biomolecular dynamics	Plasma modeling
Linear algebra	Supporting solution of partial-differential equations, fundamental benchmarks of HPL and high performance conjugate gradients	Search engine PageRank, finite-element simulations	HPC machine evaluation, climate modeling
Graph problems	Systems research, machine learning	Fraud detection	Security services, data analytics
Stochastic systems	Radiation transport, particle physics	Risk analysis in finance, nuclear reactor design, process control	Public health, modeling spread of disease

Many classes of HPC problems are designed around the supercomputer's ability to solve problems in linear algebra. In science and engineering the result of discretizing partial-differential equations frequently results in a system of linear equations. This has led to the development of both direct and iterative solution techniques for supercomputers. The main benchmark currently used to measure a supercomputer's peak performance is a dense linear algebra problem.

While many HPC problems arise from mathematical models, some of the most important supercomputing problems today arise from graph problems. Graph problems often come from problems arising in knowledge management, machine intelligence, linguistics, networks, biology, dynamical systems, and collections of pair-wise systems.

HPC problem representatives and examples of their usage in academia, industry, and government are presented in Table 1.1.

The variety and novelty of supercomputing problems continue to expand far beyond its nuclear physics roots (see the example in Fig. 1.4). As supercomputing skill-sets and resources become increasingly commonplace, it is difficult to imagine an analytical field that will not be impacted by HPC in the future.

1.1.6 APPLICATION PROGRAMMING

The principal view the user has of a HPC system is through one or more programming interfaces, which take the form of programming languages, libraries, or other services. These are expanded by

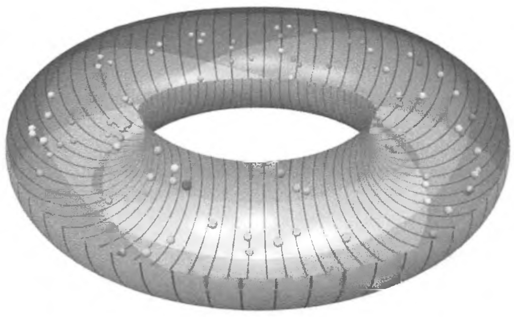


FIGURE 1.4

A particle-in-cell simulation from the Gyrokinetic Toroidal code (Princeton Plasma Physics laboratory) that simulates a plasma within a Tokomak fusion device. A sampling of some particles within the toroid is shown here colored according to their velocity, with different supercomputing processor boundaries delineated by the toroidal subdivisions.

additional sets of tools that assist in crafting, optimizing, and debugging application codes. Ironically, a major means of programming is the use of existing programs either directly or as templates to modify for specific purposes. There are hundreds of computer programming languages, from very low level including assemblers to very high reaching to the declarative regime. But for HPC the number of conventionally adopted programming interfaces is relatively few, in the order of dozens, although there are many more experimental or research models. At the risk of oversimplification, a programming language defines a set of named objects that can be manipulated, the basic operations that can be performed on these objects, the flow-control mechanisms for establishing the conditions and order of operation execution, the means of encapsulation for modularity, and I/O including mass storage.

Programming in the regime of supercomputing has additional requirements and characteristics. Performance is the driving requirement that differentiates HPC programming from other domains. It is second only to correctness and repeatability, which are of serious concern. Performance is most significantly represented by the need for representation and exploitation of computational parallelism: the ability to perform multiple tasks simultaneously. Parallel processing involves the definition of parallel tasks, establishing the criteria that determine when a task is performed, synchronization among tasks in part to coordinate sharing, and allocation to computing resources. A second aspect of programming for HPC is control of the relationship of allocations of data and tasks to the physical resources of the parallel and distributed systems. The nature of the parallelism may vary significantly depending on the form of computer system architecture targeted by the application program. Also of concern are issues of determinism, correctness, performance debugging, and performance portability.

Depending on the nature of the class of parallel system architecture, different programming models are employed. One dimension of differentiation is granularity of the parallel workflow. Very coarse-grained workloads with no interactivity, sometimes referred to as "embarrassingly parallel" or "job-stream" workflow, suggest one class of workflow managers. Fine-grained parallelism is emphasized in multiple-thread shared-memory system programming interfaces such as OpenMP and Cilk++. Medium- to coarse-grained parallelism, as reflected by highly scaled massively parallel processors (MPPs) and clusters, is primarily represented by communicating sequential processes such as the message-passing interface (MPI) and its many variants. Each of these forms of parallel programming is explored in this textbook, with extensive presentation and direct hands-on experience.

1.2 IMPACT OF SUPERCOMPUTING ON SCIENCE, SOCIETY, AND SECURITY

The broader HPC ecosystem today is a vibrant $23 billion market, projected to grow to more than $30 billion by 2020 and with a compound annual growth rate of 8%. HPC represents one of the fastest-growing markets, primarily driven by end-user demand in various application domains, including financial services, oil and gas, manufacturing, earth sciences, life sciences, national laboratories, and government intelligence.

1.2.1 CATALYZING FRAUD DETECTION AND MARKET DATA ANALYTICS

Rapidly growing global demand for financial services functions, including trading, banking, and financial transactions such as mortgage processing, is stressing financial information management systems to an unprecedented degree. Increasingly financial services companies such as proprietary trading firms, investment banks, and payment processing firms are deploying supercomputing to solve core business problems like backtesting, risk management, and fraud detection. Proprietary trading and investment management firms frequently deploy HPC systems (~100s of Tflops) to develop accurate trading strategies and predict market performance, enabling them to package highly profitable financial instruments. In many investment bank and mortgage/credit processing firms supercomputers (10–100s of Tflops) are used to process millions of records and accurately predict the risk of diverse portfolios. Payment processing firms are increasingly adopting supercomputing technologies for fraud prevention, using pattern detection and matching algorithms.

1.2.2 DISCOVERING, MANAGING, AND DISTRIBUTING OIL AND GAS

Oil and gas companies are some of the largest commercial users of supercomputing technologies, including all the publicly cited commercial petascale systems. Supercomputers drive all aspects of oil and gas workflows for exploration, production, and distribution. In exploration, supercomputers are deployed for the high-resolution seismic processing used to identify oil reservoirs through subsurface imaging (Fig. 1.5). In production workflows supercomputers are used in characterizing reservoirs and identifying the safest means of managing reserves. Increasingly oil and gas companies are using HPC capability to devise new predictive analytics for effective distribution of petroleum products. HPC is a crucial foundational capability for oil and gas companies today, and is widely deployed in exploration and production to minimize exploration risks and increase the safety of the overall processes involved.

1.2.3 ACCELERATING INNOVATION IN MANUFACTURING

Manufacturing encompasses a wide range of industries, including aerospace, automotive, consumer products, heavy industries, tire manufacturers, and electronics/semiconductor manufacturers. The common thread across these diverse industries is the use of computer-aided engineering applications for product design and manufacturing processes. In automotive industries supercomputers are used in simulating crashes, structural analysis of noise, vibration, hardness, and stress, and finally computational fluid-dynamics-driven product design. In aerospace supercomputers are primarily used

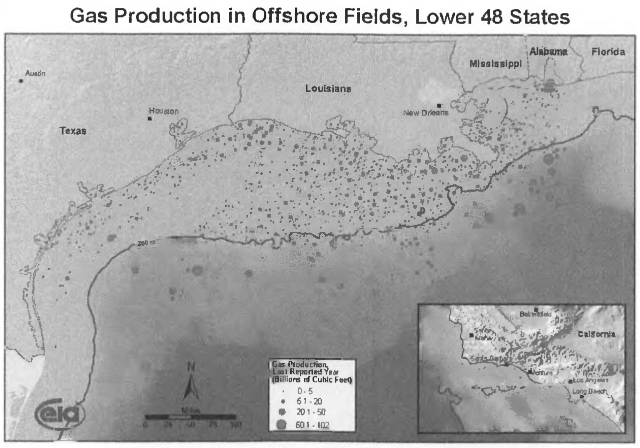


FIGURE 1.5

Researchers at BP use HPC to simulate subsurface geologies, using multidimensional analysis and characterization to identify oil reservoirs accurately.

Image by US Energy Information Administration via Wikimedia Commons

for computational fluid-dynamics-based aerodynamics simulations and virtual prototyping. By using simulation as opposed to physical testing in the design process, manufacturing firms are accelerating time to market through shortened design cycles while reducing development costs and delivering safer products to customers. HPC is perhaps one of the most important technologies in manufacturing today. Using simulation-driven engineering, organizations can improve the efficiency of jet engines, wind turbines, heavy machinery, and gas turbines (Fig. 1.6). Even a 2%–4% improvement in performance can result in billions of dollars in reduced operational and fuel costs.

1.2.4 PERSONALIZED MEDICINE AND DRUG DISCOVERY

Life sciences are another major vertical segment that relies on HPC technologies in various application areas. Supercomputing is used by researchers and enterprises for genome sequencing and drug discovery. Pharmaceutical companies often deploy supercomputers to accelerate the process of drug discovery using various molecular dynamic simulation methodologies. Using HPC and molecular dynamics simulations researchers are able to design new drugs and virtually test effectiveness, enabling significant optimization of the research process while resulting in safer and more effective drugs. HPC is also used to develop virtual models of human physiology (e.g., heart, brain, etc.), which enable scientists and researchers to understand ailments and potential treatments better (Fig. 1.7). Increasingly life sciences researchers and companies are engineering new methodologies combining genome sequencing and drug discovery to enable new and more effective forms of personalized medicine that could cure some of the most challenging diseases.

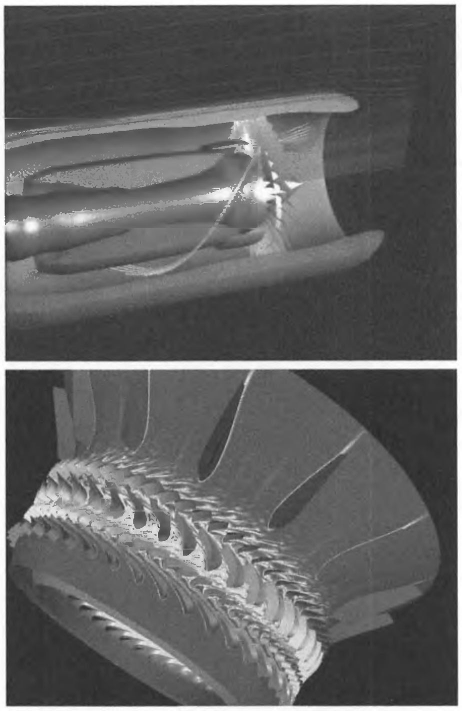


FIGURE 1.6

(Top) HPC is frequently used in high-fidelity virtual engine simulation and design. (Bottom) Researchers at NASA use HPC to simulate the design of next-generation turbines for both aviation and power production.

(Top) Simulation image via Wikimedia Commons; (Bottom) Simulation image by Dale Zante and Jay Horowitz via Wikimedia Commons

1.2.5 PREDICTING NATURAL DISASTERS AND UNDERSTANDING CLIMATE CHANGE

Another key field where HPC has delivered a transformational impact is Earth sciences. Supercomputing is frequently used to study climate change and its impact. Research organizations around the world rely on HPC to predict weather phenomena and enable highly accurate hyperlocalized forecasts. A crucial broader application area of these foundational domains is emergency preparedness, where HPC models are used to predict aspects of natural disasters such as intensity and impact of earthquakes, path and ferocity of hurricanes, direction and impact of tsunamis, and more (Fig. 1.8). The climate is ever changing, with increasing threats of intense hurricanes, heatwaves, and other

FIGURE 1.7

HPC is used to develop virtual models of kidney podocytes.

Image from C. Falkenberg et al. via Wikimedia Commons

extreme events necessitating the need for higher-fidelity computational models and more supercomputing capabilities.

In each of these application domains and beyond, supercomputers have a wide range of impact in accelerating innovation, optimizing business processes, saving lives, and delivering transformational socioeconomic impact. It is no surprise that HPC now forms a core strategic component in industrial innovation, research, and government policy development.

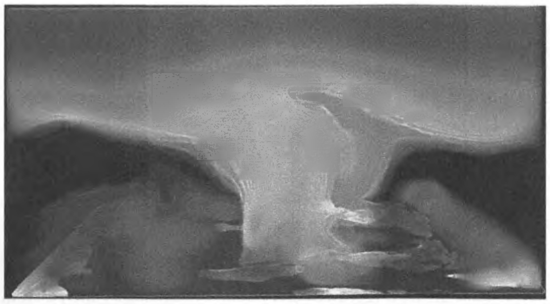


FIGURE 1.8

Researchers at Oak Ridge National Laboratory explore the advection of carbon dioxide in an atmospheric model.

Image courtesy of F. Hoffman and J. Daniel via Wikimedia Commons

Table 1.2 Broader Impacts of Supercomputing: How HPC Is Used by Different Application Domains to Accelerate Time to Innovation and Deliver Socioeconomic Impact

Vertical Segments	Common Workflows
Financial services	Fraud and anomaly detection, backtesting for algorithmic/proprietary trading, risk analytics
Oil and gas	Seismic processing, interpretation, reservoir modeling
Manufacturing	Materials simulation, structural simulations (noise/vibration/hardness and crash), aerodynamics simulations, design space exploration, thermal simulations and many more
Life sciences	Molecular dynamics, drug discovery, virtual modeling, genome sequencing and many more
Earth sciences	Atmospheric modeling, hydrodynamic modeling, ice modeling, coupled climate modeling

Table 1.2 highlights a subset of vertical segments that use HPC in their production environments to give faster time to innovation and deliver broad socioeconomic impact. Without supercomputing, these and other domains would be severely constrained in being able to deliver innovative, safer, and better products, with the pace of innovation far lower.

1.3 ANATOMY OF A SUPERCOMPUTER

To get a sense of what a modern supercomputer looks like, we give a brief description of Titan (Fig. 1.1), one of the fastest computers in the world. In November 2012 it was rated as the fastest; it has since been surpassed but is still among the top 10 supercomputers and the most powerful computer in the United States. Titan is true to its name in sheer size as well as in computational capability: it takes up more than 4000 sq ft and consumes approximately 8 MW of electrical power. Deployed at Oak Ridge National Laboratory in Tennessee, Titan was developed by Cray Inc. It incorporates the structure, function, and scale of elements found in most state-of-the-art high-end machines, with a theoretical peak performance of over 27 petaflops and delivering 17.6 petaflops R_{max} sustained performance for the HPL (Linpack) benchmark by which the Top 500 list is measured. Sponsored by the US Department of Energy and National Oceanic and Atmospheric Administration, the purpose of Titan is to do scientific research, at which it excels.

Titan is a Cray XK7 architecture: a heterogeneous architecture reflecting an important trend in HPC—the mixing of different kinds of processing units to provide the best possible operational support for distinctly different kinds of computation, even in the same application.

The block diagram of the system stack (Fig. 1.9) shows the layered hierarchy of the many physical and logical components contributing to a general-purpose supercomputer [4,5]. The bottom layer, the system hardware, represents the physical resources that are the most visible (and audible) aspect of a supercomputer like Titan. Even in this high-level view, the principal components of the system are perceived. The processors that perform the calculations and the memory which stores both the data and the program codes that operate on it are shown here, as well as the interconnection network that integrates potentially many thousands and eventually millions of such processor/memory "nodes" into a single supercomputer. Another family of hardware provides long-term storage of data and

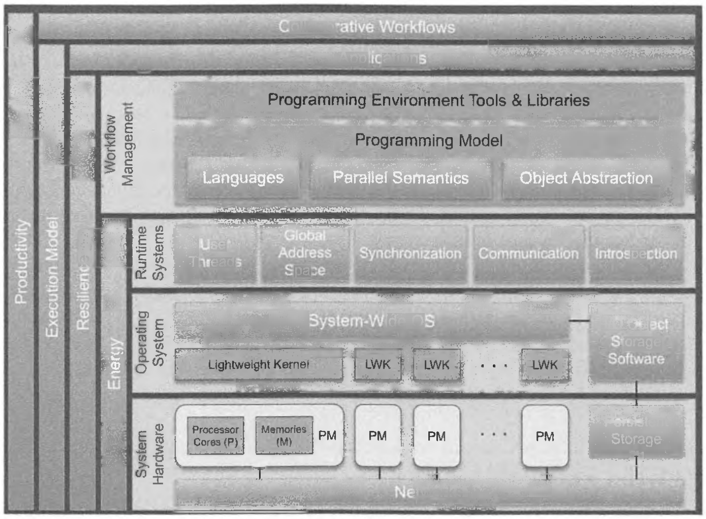


FIGURE 1.9

The system stack of a general supercomputer consists of a system hardware layer and several software layers. The first software layer is the operating system, encompassing both resource management and middleware to access input/output (I/O) channels. Higher software layers include runtime systems and workflow management.

programs. Hard-disk drives and archival tape storage can keep user data indefinitely and have much greater capacity than ephemeral main memory, but at a cost of significant access times.

The first levels of software that control the hardware and manage these resources are associated with the operating system, which is far more complex than the two layers shown would suggest. Each node has a local instance of an operating system controlling the physical memory and processor resources of the node as well as the interface to the external (to the node) system area network. An additional layer of the operating system, sometimes referred to as the middleware, logically integrates the many nodes and their local operating systems into a single system image to which users can submit their application programs and access standard I/O channels. In some supercomputers a separate front-end software environment running on a dedicated computer known as the host provides most of the user interfaces and services other than the scalable computing itself. The layered operating system reflects an abstract or virtual machine to the upper levels of the system, including the user-programming interface. It ensures a standardized set of user services upon which the application can depend, with common interface protocols independent of the specific system upon which it is performed. Among these services is the file system than manages much of the persistent storage and presents a structured organization of the diverse forms of user data.

Above the layers dedicated to resource management are those associated with the means of veloping and executing user applications and workloads. Here "workloads" are defined as a collection of loosely coupled applications, each performing a different aspect of the total set of tasks that need to be computed. For example, one might have three applications, simulator, a data analyzer, and a visualization package, with each of these three applications streaming data to the next function. The overall work management layer involves several support capabilities, including the programming languages (e.g., Fortran, C, C++), additional libraries often for parallelism (e.g., MPI, OpenMP), and compilers that provide machine-readable code for the processor cores translated and optimized from the user code. Higher-level environments and tools help in building more complex workflows and managing them. An important part of such environments are the many sophisticated libraries of previously developed and highly optimized functions that many programs can use; these enhance programmer efficiency through code reuse.

One last layer of the supercomputer system is the runtime system software. While this level tends to be a rather thin layer in conventional HPC practices, in programming models such as Java it can be much more substantial, as in the JVM (Java Virtual Machine). HPC runtime systems manage some aspects of resource management and task scheduling as well as communication. It is possible that in future supercomputers runtime systems may play a far more significant role, but such claims are speculative at this time and await confirmation through experimentation.

1.4 COMPUTER PERFORMANCE

The principal defining property and value provided by HPC is delivered performance for an end-user application. Expressions of "speed" or "how fast" are common, describing, perhaps vaguely, the relationships among time, work as computation actions, system size, and other factors. But in spite of its central role in the domain of HPC, performance itself as a measure is ambiguous, has different and in some cases contradictory meanings, and can result in different conclusions depending on interpretation. Nonetheless, despite these vagaries, performance as both an achievement and a means to achievement is core to HPC as a discipline and means of accomplishment [6]. This section briefly introduces performance as a quality metric and describes its various aspects, even as attempts are made to measure and apply it to guide system and application development.

1.4.1 PERFORMANCE

Performance is an intuitive notion of a person or a machine going well. It is a natural part of how we think about athletes, vehicles on land, sea, and air, and more abstract accomplishments such as a grade in an exam. Performance for a computer could easily be thought of as how fast it runs or the speed of an application. While not wrong, this vague idea is inadequate to allow quantitative assessment or comparisons between separate machines running the same program, separate programs designed to get the same answer, or alternative support software (e.g., two different compilers or languages) facilitating both. It is necessary to define performance in one or more useful ways and establish the quantitative metrics used to measure it. As will be seen, there are multiple meaningful and useful definitions. They share the common parameters of time (seconds) and work (primitive operations). However, depending on the context in which the two are used, the meaning can be quite different and possibly lead to conflicting conclusions.

1.4.2 PEAK PERFORMANCE

Gordon Moore, the cofounder of Intel Corporation, noted that the number of transistors in an integrated circuit doubles roughly every 2 years. This observation is widely known as "Moore's law". *Photo by the Chemical Heritage Foundation via Wikimedia Commons*

Gordon Moore is the cofounder of Intel Corporation along with Robert Noyce and a pioneer in the semiconductor industry. Gordon Moore made an observation now known as "Moore's Law" which states that the number of transistors on an integrated chip doubles roughly every 2 years. Moore's law subsequently became a driving goal in the semiconductor industry that has resulted in enormous performance gains in computational science achieved simply by adopting the latest integrated chip, an era sometimes referred to as the "free ride." Gordon Moore is a recipient of the Presidential Medal of Freedom and the IEEE Medal of Honor.

Peak performance of a system is the maximum rate at which operations can be accomplished theoretically by the hardware resources of a supercomputer. Usually in HPC peak performance is measured in units of flops (megaflops, gigaflops, teraflops, petaflops), with peak performance by the end of this decade anticipated to hit exaflops. (Note that the "s" in flops is not the plural but rather stands for "seconds". This is a common error even among otherwise knowledgeable people.) But different kinds of operations may have different peak rates. Integer operations, floating-point operations, and memory access (load—store) operations will take different times (instruction issue to instruction retire) and there will be different numbers of such operations that can be performed concurrently. To complicate this, a given operation type may take different amounts of time depending on the circumstances of the action (e.g., load operation through a cache hierarchy).

Peak performance is determined by the combination of the clock rate provided by the device technology and the hardware parallelism determined by the computer architecture. Both are a function of device density, which has demonstrated a remarkable growth rate over the last 4 decades. This trend was captured by Gordon Moore, who predicted that device density would increase by a factor of two every 2 years. This has proven uncannily accurate. Reduced feature size (i.e., width of a wire on a die) reduces capacitance and natural time constants, permitting a higher clock rate. At the same time, more devices can be put on a single semiconductor die, which permits more sophisticated processor core architectures that can do more operations per cycle (at peak). The system architecture determines the number of processor cores that together comprise the total system and contributes to the total number of operations that can be performed at the same time. Thus peak performance in terms of operations per second is determined by clock rate and architecture.

1.4.3 SUSTAINED PERFORMANCE

Sustained performance is the actual or real performance achieved by a supercomputer system in running an application program. While sustained performance cannot exceed peak performance, it can be much less and often is. Throughout the period of computation the instantaneous performance can vary, sometimes quite dramatically depending on a number of variable circumstances determined by both the system itself and the immediate requirements of the application code. Sustained performance represents the total average performance of an application derived from the total number of operations performed during the entire program execution and the time required to complete the program, sometimes referred to as "wall clock time" or "time to solution". Like peak performance, it may be represented in terms of a particular unit (kind of operation) of interest, such as floating-point operations, or it can include all types of operations available by the computing system, such as integers (of different sizes), memory load and stores, and conditionals.

Sustained performance is considered a better indicator of the true value of a supercomputer than its specified peak performance. But because it is highly sensitive to variations in the workload, comparison of different systems only has meaning if they are measured running equivalent applications. *Benchmarks* are specific programs created for this purpose. Many different benchmarks reflect different classes of problems. The Linpack or HPL benchmark is one such application used to compare supercomputers: it is widely employed and referenced, and is the baseline for the Top 500 list that tracks the fastest computers in the world (at least those so measured) on a semiannual basis.

1.4.4 SCALING

"Scaling" or alternatively "scalability" is a relationship of performance to some measure of the size (or "scale") of the HPC system. It reflects the ability to achieve increased performance for an application by employing machines of ever-greater size. Although there are many ways to quantify a system's size, a simple and widely used measure is the number of processor cores employed, recognizing that there are multiple cores per processor socket and usually multiple processor sockets per system node. The added complexity of how these are distributed in their use for a given application (e.g., how many of the cores in a given socket are actually used) is largely ignored for this purpose, although it can in fact have a significant impact on the resulting performance.

As is explored at greater depth in later chapters, an important subtlety associated with performance scaling is the application program size employed as the system size is modified. For this purpose, the size of an application can be quantified as the amount of data used, such as the dimensions of a problem matrix (n by n). Over the last 2 to 3 decades, *weak scaling* has been an important way to take advantage of ever-larger systems where the size of the data (such as n) grows proportionally with the size of the system (again, number of cores). This keeps the amount of work that a given core does about the same even as the system scale increases. Granularity of a given task (process or thread) stays about the same and so does efficiency, at least for many regular problems. This has been an important enabler of the extraordinary apparent performance gain witnessed over the last 20 years. However, as systems grow in scale the amount of main memory incorporated does not grow proportionally due to costs. As a result, the amount of memory per core has been going down, limiting the opportunity for weak scaling. At one time it was presumed that there would be about 1 byte of main memory per 1 flops of performance. Now with the largest machines this factor has shrunk to below 10% (1% for TaihuLight) in some cases.

The alternative to weak scaling is the much more challenging but important approach of *strong scaling*, where the application dataset size remains constant in the presence of increased system size. The key measure for strong scaling is not flops but time to solution. If a system were to be doubled in scale (twice as many cores), the ideal case would exhibit an execution time of half. The total work performed would be the same, but with double the number of cores it should be possible to do this in half the time. As will be seen, there are many reasons why this often is not possible, and some experts dismiss strong scaling as no longer being a viable approach. While controversial, the position taken by this discussion is that both strong and weak scaling are important, although often for different purposes.

1.4.5 PERFORMANCE DEGRADATION

The causes that result in the degradation of performance from the (not to be exceeded) peak performance to the observed sustained performance are many and varied. But they all contribute to a failure to exploit all the resources all of the time. No single part of the system is responsible for this degradation, but rather it is the imperfect match of the user application code, the compiler translation of the high-level application specification to low-level binary representation, the potential intrusion and overheads of the operating system, and the many facets of the computer architecture at the system and microcore levels. Usually, it is no single element but rather the interplay of two or more such elements that together conspire against perfection of operation. Much of this book is about how such degradation occurs and what measures can be taken to mitigate it. However, at a more abstract level four principal factors determine the delivered (sustained or actual) performance for a given application running on a target platform, here briefly introduced. This formalization of performance degradation is referred to through the acronym *SLOW*, which identifies the sources as starvation, latency, overhead, and waiting for contention.

Starvation directly relates to a critical source of performance, parallelism. Peak performance is measured with the assumption that all functional units are operating simultaneously on separate operations. If sufficient application parallel work is not available at any instance in time to support issuing instructions to all functional units every cycle, then less work will be performed than is possible, at least ideally. The achieved performance will be less than the possible peak performance. Starvation is this absence of work. Either there is not enough parallelism exhibited by the user application to keep all the system resources busy, or while there is enough work it is not distributed evenly (load balanced). In this latter case, some resources have too much work to do while other have too little.

Latency is the time it takes for information to travel from one part of a system to another. If an operation requires some data from a remote resource to proceed, latency will contribute to the amount of time it takes for that data to be delivered and cause the associated execution unit to stall or cease functioning until the data is available and it can continue. If latency for all such requests is very short, the effect is small. But if an execution unit is blocked (cannot continue) until a request from across the system is delivered, the effect can be very significant. Latency occurs in many aspects of system operation, including (but not limited to) local memory accesses, data transfer between separate nodes, and length of execution pipeline (number of stages to completion of operation). To minimize its effects on performance, latency can be reduced through exploitation of locality where everything is retained relatively close to each other. Or latency can be "hidden" by making certain that blocking of functional unit operation does not occur, even in the presence of high-latency requests. This can be achieved if a unit can temporarily work on some other task. Cache hierarchies and multithreading hardware are examples of ways of mitigating the effects of latency. But locality management by the programmer and/or the system software dominates the means of limiting latency effects.

Overhead is the amount of additional work beyond that actually required to perform the computation (such as on a pure sequential processor). Overhead work is necessary to manage resources and task scheduling, control parallelism through synchronization, support communication, handle address translation, and perform many other support functions that do not actually contribute to the operations needed for the computation itself. Overhead degrades performance through a couple of mechanisms. It wastes operations, time, and resources that are not directly associated with the computation; this alone is cause for concern and corrective measures. But there is a subtle indirect effect as well. As will be seen when we explore scaling, overhead is associated directly with setting up individual tasks, and the duration of the overhead puts a lower bound on the granularity (length) of the task it is controlling to achieve effective operation. For a fixed amount of task work (for example for strong scaling), the amount of parallelism that can be exploited is dependent in part on the fineness of granularity that can be employed and therefore the total parallelism that may be available. Thus scalability (and starvation) is indirectly affected by overhead.

Waiting of threads of action for shared resources due to *contention* of access degrades performance. When two or more requests are made at the same time to be serviced by the same single resource, either hardware or software, only one can proceed. The other(s) must wait until the first request is retired and the required resource is freed. One effect is that the delayed actions are extended in time, taking longer to complete. This has a cascading effect as follow-on actions dependent on this first one will also be extended in their time of initiation. A second effect is that the hardware upon which the delayed action is being performed may be blocked and its potential capability wasted for the duration of the delay. Both time and energy are lost. Finally, such events occur unpredictably (in most cases) and create uncertainty in the execution, making optimization methods less effective. Typical examples include bank conflicts for main memory and insufficient bandwidth for communication networks.

1.4.6 PERFORMANCE IMPROVEMENT

With these factors in consideration, the HPC user can find a number of ways to improve delivered performance—referred to sometimes as "performance debugging". This textbook continuously describes techniques to improve one's performance, including hardware scaling, parallel algorithms, performance monitoring, work and data distribution, task granularity control, and other sometimes subtle means.

Something as simple as increasing the number of nodes employed in the execution of an application can be a major method of improving performance. But one will quickly experience limitations to scaling and efficiency due to contributing factors such as uniformity (or irregularity) of tasks on processor cores, distribution of data across memory affecting bank conflicts, cache and translation lookaside buffer (TLB) misses, and page faults. Minimization of data movement, especially between system nodes, will reduce latency effects. Granularity of tasks (e.g., processes, threads) and messages will amortize overhead and latency costs if it is made more heavyweight. Exploiting compiler optimizations correctly, increasing problem dataset sizes (weak scaling), algorithm improvements, and circumventing I/O bottlenecks are among additional methods that may result in performance gain. This list of strategies is very long, if not unending. Many of these basic techniques are demonstrated in the following chapters as they relate to the respective topics.

In general, the key to performance improvement is performance measurement and profiling. Entire classes of performance measurement toolkits and frameworks have been developed directly to fulfill this important role. Due to the intricacies of performance profiling on a supercomputer, an entire chapter is dedicated to this topic later in the text.

1.5 A BRIEF HISTORY OF SUPERCOMPUTING

The history of HPC is among the most dramatic examples of human achievement through innovations in scientific discovery and engineering. Without exaggeration, no other technology in the history of mankind has exhibited such extraordinary growth and in such a narrow time span. In the course of a single human lifetime, the capability of supercomputing as measured by floating-point operation throughput has achieved a growth factor of more than 10 trillion. It is not that somehow the field just got it wrong for a period of time and then suddenly got it right. Instead, through a series of half-a-dozen epochs, device technologies, system architectures, and programming approaches, the fastest computers at any point in time grew in performance by about a factor of 200 times each decade. This unique story is easily the subject of entire volumes. But the intent here is to grasp the lessons of this history as it defines current practices in HPC and guides future developments in to the era of exascale computing. Examples from the history of supercomputing illustrate and highlight critical ideas, and their importance in defining HPC and driving its progress. The overall pattern of the history of HPC provides a framework for future detailed discussions. But even in this initial presentation the fundamental concepts, touched on briefly above, are engaged as the fabric from which the more precise patterns of this story later evolve.

The age of modern computing has been powered by continuous and significant technology advances and achieved through innovations in computer architecture and programming models. But this accomplishment has only had true meaning and value due to the driving needs and empowerment of end-user goals: specifically domains of consideration that could only be addressed by computational means. The conceptual underpinnings of this constructive tension have been the continuous need to increase speed and concurrency while minimizing sources of inefficiencies. The changes in architecture were driven by a need to exploit the new opportunities enabled by new emergent technologies while addressing the challenges that each new advance imposed. Thus one perspective (there are many) of the history of supercomputing is to examine briefly seven epochs enabled through device technology and achieved through computer architecture supported with responsive programming models.

1.5.1 EPOCH I—AUTOMATED CALCULATORS THROUGH MECHANICAL TECHNOLOGIES

Vannevar Bush with US President Harry Truman. *Photo by Abbie Rowe via Wikimedia Commons*

Vannevar Bush's pioneering work on large-scale analog computers for solving differential equations paved the way for the tremendous expansion in computational science that continues to grow this day. The "differential analyzer", named and developed by Bush while at Massachusetts Institute of Technology (MIT), became the first general-purpose, large-scale analog computer for integration and served as a catalyst for the subsequent development of electronic digital computers. Apart from his influential impact on government support for big science and wartime management of scientific research, Bush is also considered an internet pioneer even though he had no direct involvement in the development of the internet. His influential 1945 essay describing a theoretical device that could store and access documents through associative linking was credited by information technology pioneer Theodor (Ted) Nelson with helping inspire the hypertext concept used in the internet today.

For multiple millennia mankind has sought aid in performing calculations through mechanical means, both to store intermediate values and to perform arithmetic operations on them. These methods increased speed and improved accuracy compared to pure mental or written methods. Although primitive, these techniques proved very effective and supported many important tasks associated with commerce, logistics, and even early science. Enumeration and summation were among the earliest computing tasks and these were facilitated by a family of simple recording media, such as the "tally sticks" used more than 10,000 years ago, culminating in the "abacus" in its many forms in Babylonia at about 2400 BCE. The abacus permitted integer numbers to be represented through physical arrangements of beads, and for addition and subtraction operations to be performed through mechanical actions. By 200 BCE the Chinese had developed the suanpan, an advanced form of abacus that has been employed up to the modern era in some parts of the world. This achieved two principles that are relevant to today's supercomputers: artificial representation and storage of numbers and the concept of process, a sequence of simple actions to achieve a more complex result. A third aspect of modern digital computation is manifest: the discrete data representation referred to now as "digital".

The 17th century and the age of enlightenment saw the first developments of the mechanical calculator. Blaise Pascal, a French mathematician, invented the "Pascaline" in 1642; this simplified the user interface such that anyone could use it and incorporated a carry mechanism for addition and subtraction. Gottfried Leibniz developed a "stepped reckoner" that performed the operation of digital multiplication in 1671. Throughout the 18th century many advances were made in mechanical calculators, culminating in 1820 with the "Arithmometer" developed by Charles de Colmar, which was mass produced from 1851 and performed division, addition, subtraction, and multiplication. This sequence of developments over a period of 2 centuries achieved practical artificial calculation delivered in a useful form to a wide market. It also provided a functional capability that would be embedded in all future modern computers, now referred to the "ALU" or "arithmetic logic unit".

At the beginning of the 19th century the second major advance toward fully automated computing was introduced by Joseph Jacquard for the special-purpose application of weaving. His contribution in 1801 was the concept of sequence of control through the storage of commands or instructions capable of eliciting specific automated action. The "punched card" was invented, which through a pattern of holes could define and activate a particular action of a possible set of many. The "Jacquard loom" was fed a sequence of such cards determining the color and pattern of weaving of threads to produce cloth. In 1890 the punched card storing data (rather than instructions) and the mechanical calculator were integrated to form the "tabulator", developed by Herman Hollerith and first applied to the US census. This provided the foundation for complex data processing by mechanical means that dominated commercial information management for almost a century and established what would become the world's largest computer company, IBM (originally the Tabulating Machine Company).

The basic concepts for general fully automated calculation by mechanical means were established by Charles Babbage, a British mathematician, starting in 1834 with his design of the "analytical engine" [7]. His earlier ideas of the "difference engine" which computed polynomial tables and was eventually built led to conceptual extensions for general-purpose computation. It embodied the principles of the mechanical ALU with that of punched-card sequence control. Babbage was not to see the realization of this dream, but he influenced the work of Konrad Zuse, who in 1938 completed the first programmable mechanical computer in Germany, and in 1944 Howard Aiken and IBM developed the "Harvard Mark 1". This final system was the culmination of 3 centuries of advances, starting with the Pascaline, that demonstrated the viability of automated computation, including the basic concepts of arithmetic function units, sequence control through stored instructions, intermediate data storage, and I/O. But despite all these innovations, the resulting calculating speed, i.e., the rate at which operations could be performed, was relatively slow: about 1 instruction per second (IPS). It would take a breakthrough in device technology, unanticipated in the time of Pascal and Leibniz, as well as further conceptual advances beyond that of Babbage to create the paradigm shift that led to the modern digital computer and supercomputer.

1.5.2 EPOCH II—VON NEUMANN ARCHITECTURE IN VACUUM TUBES

The second epoch of high-speed computing (what we would call "supercomputing") incorporated a paradigm shift that was enabled by a component technology revolution and driven by the crisis of war. Four fundamental concepts laid the groundwork for the modern computational age: Boolean logic, binary arithmetic, computability, and what would be called the von Neumann architecture. Logic was developed by the British mathematician George Boole in 1848 and provided the fundamental framework for the design of complex digital logic functions through the synthesis of basic Boolean operations (e.g., AND, OR, NOT), the basis of virtually all modern computers today. In 1937 Claude Shannon derived the basic unit of information, the "bit" (binary digit) that comprises the basis for binary arithmetic, the principal means of conducting calculations by artificial means (decimal or base 10 had also been used). A year earlier, in 1936, Alan Turing presented what would become the fundamental model of computation, the "Turing machine", which resolved a key issue of computability put forth by Hilbert and also addressed work by Church. Today we determine that a computing machine is general purpose if it is a "Turing equivalent". Finally, the mathematician John von Neumann, influenced by the work of Eckhart and Mauchly, described a class of general-purpose stored-program digital computing that has served as the basis of the architecture for almost all central processing unit (CPU) designs to this day. Central to this was the concept of the program counter and program representation as a sequence of encoded instructions stored in the main memory where the data also resides.

Alan M. Turing

June 23, 1912–June 7, 1954

Alan Turing laid the theoretic foundations of what would become the field of computer science, and in so doing defined what computability itself was. He applied his concepts and insights into the development of automated computing systems for breaking the German Enigma code during World War II, and developed one of the first stored-program digital electronic computers, the automatic computing engine (ACE). Due to his theoretical and applied research, Alan Turing is noted as one of the premier contributors to the establishment of modern computing.

Turing devised the concept of an abstract computing construct that would later come to be called the "universal Turing machine", and was employed to solve a central mathematical problem posed by the mathematician Hilbert, known as the halting problem. This established the principles of computability (also contributed to by Church with an alternate abstraction, the lambda calculus). Through this work a deep understanding of how all computers must work was devised, as well as the basic idea of computer algorithms. Any computer that can be fully general purpose is identified as a "Turing equivalent" to this day.

Turing brought his prodigious acumen and insights in mechanical computation to the requirements of crypto-analysis, and more specifically code breaking at Bletchley Park in England, where the challenge of decoding Enigma messages was a priority. His contributions to the development of advanced "bombes", electromechanical computers that tested many possible decoding combinations, are considered critical to bringing World War II to a successful and relatively fast conclusion and saving many lives. Among other effects, it was crucial to helping win the Battle of the Atlantic, enabling supplies and personnel to cross the ocean from the United States and Canada to Great Britain.

Alan Turing produced the first full design of a stored-program digital electronic computer, the ACE, which for various reasons was not built until after his premature death. However, a smaller version, the Pilot ACE, was implemented and the concepts incorporated in this early work had significant impact on future computer designs. Toward the end of his life, Turing considered the implications of mechanical computing and the idea of artificial intelligence. He conceived of a test that would mark the achievement of a thinking machine; this has come to be called the "Turing test" for machine intelligence.

—CONT'D

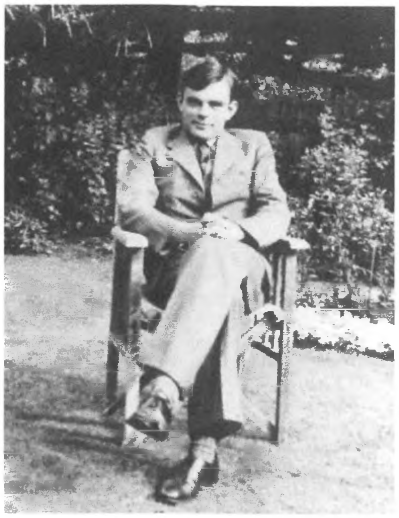


Photo via Wikimedia Commons

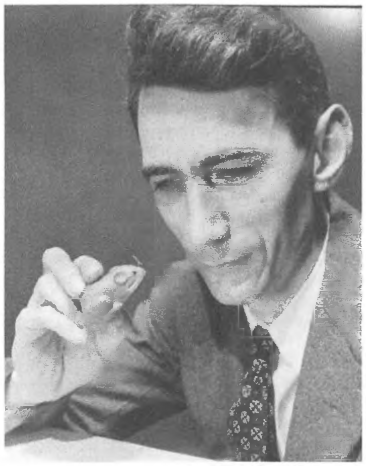


Photo from Wikimedia Commons

Claude Elwood Shannon is widely considered to be the father of modern information theory. He applied probability theory to the communication problem of finding the best way of encoding information for transmission. One of the fundamental

–CONT'D

concepts of this field, information entropy, describes the measure of information contents in a message. The unit of entropy has been named the "shannon" in his honor, and is equivalent to 1 bit.

Shannon created one of the first digital logic circuits based on the work of George Boole. At that time such circuits used arrangements of electromechanical relays, each of which could be in one of two states: "on" or "off". This corresponded to the 0 and 1 values used by binary Boolean algebras. Shannon expanded the concept by rigorously proving that digital circuits are capable of solving Boolean algebra problems, thus effectively establishing the theoretical foundations of digital logic design.

Shannon's research at Bell Laboratories focused on cryptography and weapons control systems during World War II. He proved that cryptographic one-time pads are unbreakable. He is credited with the invention of signal-flow graphs that may be used to describe signal propagation in a broad class of cyber-physical systems. He also introduced sampling theory that analyzes the relationships between the continuous-time (analog) signal and its uniformly sampled discrete-time representation. One outcome of this research is the famous Shannon–Nyquist theorem that quantifies the impact of signal aliasing. After leaving Bell Laboratories Shannon joined MIT, where he taught and worked at the Research Laboratory of Electronics until 1978.

During his prolific career Shannon maintained professional contacts with Alan Turing, Hermann Weyl, John von Neumann, Hendrik Bode, John Tukey, and many others. His achievements were recognized by numerous awards, including the Morris Liebmann Memorial Prize, Stuart Ballantine Medal of the Franklin Institute, AIEE Mervin J. Kelly Award, National Medal of Science, IEEE Medal of Honor, Royal Netherlands Academy of Arts and Sciences Joseph Jacquard Award, Audio Engineering Society Gold Medal, Kyoto Prize, and Eduard Rhein Foundation Basic Research Award, as well as multiple honorary doctorates bestowed by institutions of higher learning around the world. He was also inducted into the National Inventors' Hall of Fame in 2004.

MAURICE WILKES AND THE ELECTRONIC DELAYED STORAGE AUTOMATIC CALCULATOR

Maurice Wilkes was a pioneer in and major founder of the field of digital electronic computing. Arguably he is responsible for the first practical stored-program digital computer, the electronic delayed-storage automatic calculator (EDSAC), launched in May 1949. Wilkes spent his professional life primarily at the University of Cambridge as both a student and a professor, ultimately becoming director of the University of Cambridge Computer Laboratory. Throughout his career Professor Wilkes made a number of formative advances in computer principles and practices that led the field from its inchoate phase to commercial success; among these were EDSAC, microcontrollers, and microprogramming. His influence is felt even today after his death in 2010 at the age of 95.

EDSAC was derived from the concepts of von Neumann, Eckert, and Mauchly and from his in-depth experience with the electronics technology of the 1940s, primarily in radar during World War II. This included vacuum tube devices and circuits, mercury delay lines (tanks) for short-term data storage, and pulse transformers for data communication. Electromechanical devices used for tabulation and business data formation, search, and enumeration were employed for data entry and long-term storage on paper tape with teleprinters for output. The tanks held 1024 (initially 512) words, each of 18 bits. This was an accumulator machine with an additional buffer to aid in multiplication. It was a conservative machine, in that its clock rate was only about 660 Hz (1.5 ms), but this delivered the important reliability required for practical application computations. Double words of 35 bits could be used, and the accumulator was 71 bits long to hold two of these. The binary instruction set included a 5-bit op code, a 10-bit operand (often an address), a 1-bit length code, and a spare bit.

It was Maurice Wilkes who introduced the concept of microcontrollers to computer design. A microcontroller is a computer within a computer optimized to generate the control signals for the computer operations efficiently. By changing the microcode in the control store (usually a fast memory), the computer's instruction set could be added to or improved without changing the hardware. This solution was inspired by Wilkes's visit to MIT, where the Whirlwind computer was being developed using a similar hardwired controller with a sequence of rows of diodes determining the order of control signals. It was Wilkes's innovation to change the diodes to electronic switches. EDSAC-2 would later be the first computer to be microcoded, setting a trend that lasted at least 25 years.

MAURICE WILKES AND THE ELECTRONIC DELAYED STORAGE AUTOMATIC CALCULATOR–CONT'D

Maurice Wilkes and the EDSAC I under construction. *Copyright Computer Laboratory, University of Cambridge. Reproduced by permission*

John von Neuman. *Photo by Los Alamos National Laboratory via Wikimedia Commons*

John von Neumann's enormously wide-ranging scientific and mathematical work includes significant contributions to computer architecture, supercomputing algorithms, and cellular automata. His name is commonly associated with the

–CONT'D

stored-program architecture used by modern computers, even though the stored-program concept is also attributable to the developers of the electronic discrete variable automatic computer, J. Presper Eckert and John William Mauchly. John von Neumann's work on the electronic numerical integrator and computer (ENIAC) led to his realization that it could compute pseudorandom numbers, thereby paving the way for the first Monte Carlo simulations ever performed on an electronic computer in 1948. von Neumann was also involved in the first numerical weather predictions, also using the ENIAC. Both Monte Carlo simulations and numerical weather predictions are mainstays on supercomputers today.

The revolutionary enabling device technology that inaugurated modern computing was electronics, the amplification and control of electricity through the agent of active device components, the first of which was the vacuum tube. The vacuum tube was produced by the prolific inventor Thomas Edison by accident while working on the electric light in 1880. He witnessed the counterintuitive phenomenon of an electric current flowing in a vacuum between two disconnected elements (cathode and anode). Called the "Edison effect" (he patented it), it did little more than rectify an alternating current but served as the basic technology upon which electronics would be devised. The second breakthrough was the use of the vacuum tube as an amplifier by adding one or more screens between the cathode and the anode to which a much weaker signal could be applied. The Audion, the first amplifying vacuum tube, was produced by Lee De Forrest in 1906 and led to a series of advances, starting with the "Triode" that allowed a strong current to be controlled by the much weaker one. In 1937 John Atanosoff developed digital logic circuits for binary computing using vacuum tubes as electronic switches, replacing the mechanical counterparts employed by Zuse and Aiken and all of their forerunners.

During World War II Eckert and Mauchly in the United States and Turing in Great Britain developed special-purpose vacuum-tube-based digital electronic calculating systems for ballistics calculations (ENIAC) [8] and code breaking (Colossus), respectively. Immediately after the war, the von Neumann architecture concept was applied as the basis of the first generation of the modern digital computer for several projects in the US, England, and Germany. EDSAC was developed at Cambridge University by Maurice Wilkes in 1949 as the first full embodiment of the combination of principles and technologies described above [9]. In the United States a number of machines were implemented, including the MIT Whirlwind, the IBM 704, the IAS, and the UNIVAC I among others. Of these, Whirlwind was perhaps the supercomputer of the day. Performance of these systems ranged from below 1 KIPS to about 10 KIPS, and was principally limited by the speed of data-storage technology. After early flirtations with such primitive storage technologies as mercury delay lines and Williams' tubes, memory using small toroidal ferromagnetic cores to store magnetic fields was developed as part of the Whirlwind project and mass marketed by IBM to provide a stable, dense (relatively), three-dimensional (3D) matrix of memory bits that revolutionized digital computing and served as its main memory for more than 2 decades.

The first commercially produced computer was the LEO 1 developed by J. Lyons & Co. in the United Kingdom, based on EDSAC. It was capable of approximately 600 IPS. The first commercial digital electronic computer in the United States was the UNIVAC I, delivered in 1951 by Remington Rand. IBM produced its first scientific computer, the 701, and first mass-produced commercial machine, the 650, in the mid-1950s. The 701 was capable of 4 KIPS.

Performance for this first generation of digital electronic computers was a direct function of the clock rate of the CPU and the number of bits processed in parallel. Clock speed and parallelism would become the two principal dimensions driving the evolution of supercomputer performance. It was also dependent on the number of clock cycles required to perform each operation.

1.5.3 EPOCH III—INSTRUCTION-LEVEL PARALLELISM

The next technology breakthrough to revolutionize supercomputing was the development of the "transistor" in 1947 at Bell Laboratories. The transistor served as an alternative to the vacuum tube as an electronic switch. Unlike the tube, the transistor employed semiconductor materials to control the flow of electrons through a solid medium. It was much smaller and faster than the vacuum tube, required much less energy, and was more reliable. Eventually it became much more cost effective as well. Without exaggeration, the transistor made the digital computer practical and commercially viable to a large market, guaranteeing its positioning as a strategic technology. As a functional device, the transistor had three connections: emitter, collector, and base. These, roughly compared to the older vacuum tubes' cathode, anode, and grid in their respective roles, with the weak current into the base being amplified as a much larger current between the emitter and collector. The transistor went through two phases of evolution: the first using germanium, and the second employing silicon. The latter demonstrated significant improvements over germanium, and once perfected largely replaced it. These bipolar transistors were themselves replaced with field-effect transistors that greatly increased input impedance, providing lower current drains between successive circuit stages and superior isolation between them for improved operational properties and easier circuit design [10].

The early transistor-based computers were first and foremost advances in circuit design, replacing the previous vacuum tube epoch with new logic circuits using transistors. Among the first experimental machines was the TX-0 developed by Lincoln Laboratories, demonstrating in principle the viability of transistor digital electronics for stored-program digital computing. As transistors improved in quality and reliability, design practices became standardized and printed circuit boards and modules were developed that reflected the higher abstraction of logical gates (Boolean functions) and latches (single-bit storage) with well-defined voltage levels representing binary 1 and 0 or Boolean true and false. From these, computer architectures were constructed. The IBM 1401 launched in 1959 (of which more than 10,000 were delivered in various configurations) and the DEC PDP-1 launched in 1960 (which began the minicomputer) and based on the TX-0 were two among many. They all executed one instruction at a time. The IBM 7090, also launched in 1959, was essentially a transistor version of the previous vacuum tube 709 but six times faster.

While every advance in delivered performance may be interpreted as improvement in HPC, it became clear even in the first generation of digital electronics that system design and programming for business purposes and scientific applications were different, with the former emphasizing long-term data storage and I/O devices while the latter required optimization for numeric computation with an emphasis on floating-point (real numbers) operations. As basic methods of circuit design, reliability, and cost were refined, these two distinct domains of computing began to be distinguished as two increasingly disparate system designs or architectures. Ultimately a machine architecture emerged which so dramatically reflected this demarcation of combined purpose-built design that it would be recognized as the first true supercomputer. That machine was the CDC 6600.

Developed under the leadership of Seymour Cray and designed by Jim Thornton, the CDC 6600 was delivered in 1964 by Control Data Corporation (CDC), and in its various forms was deployed at over 100 user sites. It was the first 1 megaflops (peak) supercomputer. While a *tour de force* in the use of the new silicon transistor technology that enabled it, providing an unprecedented 10 MHz clock rate, it was its use of innovative computer architecture to provide and exploit lightweight or instruction-level parallelism (ILP) that catalyzed a revolution in HPC, here categorized as the third epoch, arguably creating the supercomputer itself. It incorporated 10 separate logic units and was served by as many peripheral processors for accessing memory and I/O channels. Each was served in its turn by the CPU, overlapping the many operations and for the first time adding this new level of parallelism to achieve a dramatic performance increase.

1.5.4 EPOCH IV—VECTOR PROCESSING AND INTEGRATION

The SX-9. *Photo by GenGen via Wikimedia Commons*

Tadashi Watanabe is a computer architect and engineer, primarily responsible for the design of the highly successful SX series of vector supercomputers. The NEC SX-2, introduced in 1983, was the first machine to break the 1 GFlops barrier. It utilized four sets of pipelines feeding 16 vector arithmetic units implemented in high-density large-scale integration (LSI) logic and operating at a 6 ns machine cycle. It was also the first Japanese liquid-cooled supercomputer. Over the years the SX series significantly expanded in memory capacity and computational throughput while lowering the energy requirements, including, for example, the first single-image multinode installation in 1994 that performed at 1 TFlops peak utilizing 512 processors housed in 16 nodes, marketed as the SX-4. Later, the SX-6 served as the building block of the famous Earth Simulator that was used to perform the unprecedented whole-earth climate simulations at 10 km grid resolution. The best-performing member of the family, the SX-9, scales to 512 nodes that comprise a grand total of 8192 processors and 512 TB of memory, achieving in aggregate 839 TFlops. While at Riken R&D Center, Watanabe also influenced the design of another large-scale computer, K, that debuted at number one position in the Top 500 list in 2011, delivering over 8 PFlops in Linpack, and 6 months later was the first machine to cross the 10 PFlops threshold. For his achievements Tadashi Watanabe was honored with the IEEE-ACM Computer Society Eckert-Mauchly Award, the IEEE Computer Society Seymour Cray Award, the US National Academy of Engineering (Foreign Associate) Prize, and the Japan Academy Prize.

Seymour Cray and the Cray-1

September 28, 1925—October 5, 1996.

If any one person can be described as "the father of supercomputing", Seymour Cray is that person. Cray was largely responsible for the first true supercomputer while at CDC, the CDC-6600, delivered in 1965 with a performance well in excess of 1 megaflops on real-world applications. More than 100 of the 6600s were sold to national and academic laboratories. It was superseded in its status as "fastest computer in the world" in 1969 by the CDC-7600, also designed by Cray, and 10 times faster than the 6600 with a peak performance of about 36 Mflops. An attempt at a third-generation supercomputer, the CDC-8600, failed due to cost overruns.

Seymour Cray founded a new company, Cray Research Inc. (CRI), to chart the future direction in supercomputers, resulting in the Cray-1, the first true vector computer, launched in 1976 and beating just about anything on the market to a significant degree with a peak performance of 100 Mflops and more than 80 systems shipped. This was a really big success, and as much as any single machine defined supercomputing. The Cray-1 had both exceptional vector throughput and scalar speed.

Other teams at CRI extended the Cray-1 to the Cray-XMP and the Cray-YMP by employing multiple vector pipelines while improving clock rates and memory speeds. Seymour Cray developed the Cray-2, which was highly innovative but due to delays in delivery and poorer than expected memory speed technology failed to achieve a large market share. The design of the Cray-3 and Cray-4 by his new startup, Cray Computer Corporation (CCC), proved unsuccessful, with the ultimate bankruptcy of CCC due to heavy investments in gallium arsenide technology and the emergence of MPPs using very large-scale integration (VLSI) complementary metal-oxide semiconductor technologies which delivered superior performance to cost. Sadly, Cray died in a car accident in 1996 shortly after the founding of his last company, SRC.

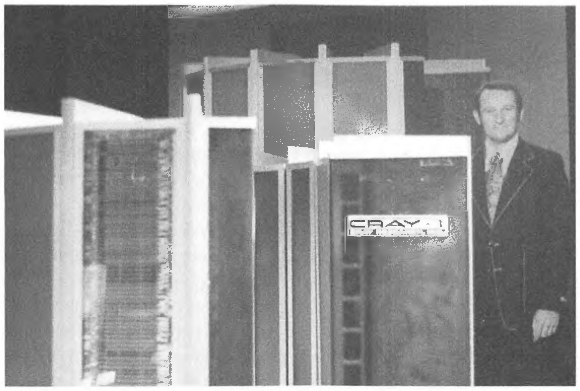


Photo by Michael Hicks via Wikimedia Commons

As silicon transistor technology matured and feature size shrank it became possible to integrate more than one transistor along with diodes, resistors, and connecting wires on a single semiconductor die. The integrated circuit emerging in the late 1960s repeated the breakthrough of the original transistor, again pushing size, speed, power, and cost to new and unprecedented levels and driving the next revolution in supercomputing. This technology in its earliest phase was referred to as "SSI" for "small-scale integration" and incorporated one or more logical gates on a single die, exposing all their inputs and outputs via pins. SSI was formed in a number of basic technologies, such as resistor

FIGURE 1.10

The Cray-1 Supercomputer. First deployed in 1976 with a peak performance of 136 MFlops, this machine inaugurated the modern age of supercomputing.

Photo by Clemens Pfeiffer via Wikimedia Commons

transistor logic, diode transistor logic, transistor logic, and ECL (emitter coupled logic), among others. ECL was probably the fastest integrated logic at the time, but consumed more power.

As before, technology enabled significant advances in clock rate, but more importantly it inspired the next paradigm shift in supercomputer architecture: the exploitation of pipelining structures to process vectors of numbers. In 1976 Seymour Cray delivered the Cray-1 supercomputer to Los Alamos National Laboratory (Fig. 1.10) [11,12]. This architecture exploited the new technologies to achieve the then high clock rate of 80 MHz and vector pipelined processing for ultralightweight parallelism. But the pipelining did not just contribute to exposing and exploiting new levels of parallelism; it was also necessary to permit the 12.5 ns cycle time to be achievable by reducing the amount of logic that had to be processed at each physical point in the system. Further, the Cray-1 through this form of architecture addressed key factors of efficiency, specifically latency and overhead.

The key idea to pipelining is to divide a function into a balanced set of successive subfunctions, each of which takes much less time than performing the full function at one time. Each subfunction or pipeline stage operates simultaneously with the others but on different sets of operand data. Thus for a p-stage function pipeline, *p* different operations are being performed simultaneously, but at different stages of their completion. At the completion of a compute cycle, all the intermediate results of each stage are passed to their successive stage. A new set of operands is fed to the first stage of the pipeline, while the final result values are extracted from the final stage. The set of operand values forms a vector

—CONT'D

a payload bandwidth of approximately 500 MB/s per link. This permitted efficient remote memory access with storage capacities ranging from 64 MB to 2 GB per node. One of the machine's unique features was self-hosting: no additional front-end system was required to manage its operation. A 1480-processor T3E system was the first to achieve a teraflops performance in a scientific application (simulation of metallic magnetism) in 1998. The Cray X1 combined the advanced network of the T3E with improved memory bandwidth and vector performance of up to 12.8 GFlops per processor to deliver over 50 TFlops aggregate in the largest configuration. The follow-on, the "Black Widow", introduced four-way symmetric multiprocessing nodes and a high-radix (64-port) yet another router chip that permitted growing the system to 32K processors in a fat-tree topology with the worst-case diameter of 7 hops.

While Scott's career was closely associated with CRI and Cray Inc., he also served as senior vice-president and chief technology officer of the Tesla business unit at Nvidia and principal engineer of the Platforms Group at Google. He holds 27 patents on interconnect, processor, and cache architecture, synchronization mechanisms, and parallel processing.

The previous epoch in HPC was triggered by the advent of VLSI technology, which has ultimately resulted in literally billions of transistors on a single semiconductor die. This technology demanded new strategies for making best use of the enormous capability now possible. It also opened up a new relationship between the special needs of supercomputing and the mass-market needs of general computing that was also enabled through VLSI. This was the era of the "killer micro".

VLSI permitted more concentration of functional ability on a single chip than ever before. This was most dramatically reflected by the microprocessor, a logical element with all the necessary functionality to perform complex computations and handle the workload of user application programs. Where once such a machine would cost a million dollars and fill a large machine room, by the beginning of this epoch a deskside or desktop box could do the same work and cost less than $40,000. Unlike the other epoch-spanning technologies, VLSI itself went through orders-of-magnitude transitions in device density throughout its 2-decade duration. Prior to this period, early microprocessors provided basic functionality with very limited performance. The first microprocessors actually entered the market in the 1970s, with 4-bit and 8-bit microprocessors being used in first-generation personal computers (PCs) and 16-bit microprocessors available at the start of the 1980s, causing a mitosis of the market between lower-cost PCs for personal use and higher-cost "workstations" for industrial-grade purposes. Early experiments were conducted in the late 1980s to explore the potential of integrating multiple microprocessors into ensemble systems, including the Caltech Cosmic Cube, MIT Concert, IBM RP2, Intel Touchstone Delta (Fig. 1.12), and others. In the meantime, farms of workstations on local area networks sharing I/O devices such as printers, early mass-storage systems, and access to the precursors to the internet were pursued as a means for cycle harvesting to perform large workloads on systems not in use (such as idle workstations at night).

By the beginning of the 1990s the first commercial MPPs with custom networks were being offered by vendors. Among these were the Intel Touchstone Paragon (1994), the Thinking Machines Corporation CM-5 (1992), and the IBM SP-2. With distributed-memory hardware a new model of programming was required, one in which each processor performed a separate process. Coordinated action was achieved through a combination of data exchange using message-passing methods and synchronization primitives, both across the interconnection network. CRI also introduced the T3D and later the T3E that integrated microprocessors, but in a configuration that permitted a degree of shared

FIGURE 1.12

Intel Touchstone Delta. With over 500 cores connected with a mesh topology, it delivered performance of 10–20 GFlops.

Photo courtesy of Dr. Paul Messina

memory across the entire system. Silicon Graphics, Inc and Convex extended the level of shared memory to include a nonuniform memory-access cache-coherent model through hardware.

A second strategy in the exploitation of VLSI microprocessor and high-density DRAM technologies emerged as a result of the success of the commercial market for workstations and the dramatically larger consumer market for PCs. The commodity cluster is a form of high performance computer assembled from commercially manufactured subsystems, each of which serves its own market niche as a standalone product. The cluster "node" is a computer that can be directly employed individually as a workstation, PC, desktop, or deskside machine, or as part of a set of independent computing facilities. Originally the network was derived from technology used as local area networks. In each case the markets for individual components greatly exceed the market for the components in cluster functions, enabling economy of scale to increase performance to cost dramatically with respect to custom-designed MPPs of the same scale (i.e., number of processor cores).

The concept of clustering of computing units predates this epoch, with perhaps the earliest example in the 1950s in the development and deployment of SAGE, a multiple computer system produced by IBM for North American Aerospace Defense Command to defend North America from the threat of air attack. The term "cluster" was first adopted by the DEC M31 Project (Andromeda), which assembled 32 VAX 11/750 minicomputers into a single ensemble system in the late 1980s. A number of early projects were initiated to investigate the feasibility and utility of harnessing the aggregate power of clustered systems, most notably the UC Berkeley Network of Workstations (NOWs) and the NASA Beowulf Project, both started in 1993. NOWs devised a series of increasingly sophisticated clusters of workstations, stressing the importance of the highest-quality and highest-performing components. In 1997 the first commodity cluster to be represented on the Top 500 list was the Berkeley NOW. The Beowulf Project pursued an alternative strategy, exploiting mass-market consumer-grade components to achieve the best possible performance to cost, even at the expense of efficiency and performance. Beowulf was also the first application of the inchoate Linux operating system for scientific parallel computing, to which the project contributed a large number of network drivers [14,15]. As this epoch comes to an end, the formula of commodity clusters now dominates the Top 500, with more than 82% of deployed systems in this category. The Beowulf integration of clusters of x86 processor architectures, the family of Ethernet networks, the Linux operating system, and message-passing programming modes all dominate in their respective categories the field of HPC due to the contributions of many researchers and developers in the field.

1.5.7 EPOCH VII—MULTICORE PETAFLOPS

It is controversial to assert that HPC is departing from the epoch of communicating sequential processes and being driven by technology toward something else in the petaflops era. But the now ubiquitous use of multicore sockets and graphics processing unit accelerators combined with exploration and experimentation of hybrid programming methods strongly suggests that the field is in a phase transition, with the ultimate outcome still undetermined [16]. Performance gains like never before are now determined by the growing number of cores employed, while programming models and methods are struggling to catch up. But some applications are failing to take full advantage of the hardware resources available, and are thus being dropped by the wayside as ever-fewer programs operate effectively across all systems.

A dominant trend is toward the synthesis of coarse-grained distributed-memory techniques using independent processes and medium-grained shared-memory techniques using interrelated multiple threads. Also being pursued is the addition of or replacement by lightweight processing cores for greater control state and higher memory-usage bandwidth. Nonetheless this is a rapidly evolving area, with large-scale system architectures evolving at least incrementally and programming methods changing to support them.

Even as this strategy is pursued, alternative pathfinding techniques are being explored to take advantage of dynamic adaptive computing methods supported by a new generation of runtime system software and programming interfaces. The future is far from known in this regard, but what will emerge when the smoke clears will be as interesting and exciting as any of the prior epochs.

1.5.8 NEODIGITAL AGE AND BEYOND MOORE'S LAW

The international HPC development community will extend many-core heterogeneous system technologies, architectures, system software, and programming methods from the petaflops generation to exascale in the early part of the next decade. But the semiconductor fabrication trends that have driven the exponential growth of device density and peak performance are coming to an end as feature size approaches nanoscale (approximately 5 nm). This is often referred to as the "end of Moore's law". This does not mean that system performance will also stop growing, but that the means of achieving it will rely on other innovations through alternative device technologies, architectures, and even paradigms. The exact forms these advances will take are unknown at this time, but exploratory research suggests several promising directions—some based on new ways of using refined semiconductor devices, and other complete paradigm shifts based on alternative methods. Other forms will be incremental changes to current practices benefiting from a legacy of experience and application.

While not commonly employed, the term "neodigital age" designates and describes new families of architectures that, while still building on semiconductor device technologies, go beyond the von Neumann derivative architectures that have dominated HPC throughout the last 6 decades and adopt alternative architectures to make better use of existing technologies. The von Neumann architecture emphasizes the importance of arithmetic floating-point units (FPUs) as precious resources which the remainder of the chip logic and storage is designed to support. It also enforces sequential instruction issue for execution control. Complexity of design offers many workarounds, but the fundamental principles prevail. Now FPUs are among the lowest-cost items and parallel control state is essential for scalability. New advances to current architecture and possible alternatives to von Neumann architectures may be among the innovations to extend the performance of semiconductor technologies beyond exascale.

More radical concepts are being pursued, at least for certain classes of computation. Special-purpose architectures where the logic design and dataflow communications match the algorithms can significantly accelerate computations for specific problems. Digital signal processing special-purpose chips have been employed since at least the 1970s. More recently architectures such as the Anton expand the domain of special-purpose devices to simulation of N-body problems, principally for molecular dynamics. Even more revolutionary approaches to computing are targets of research, including such techniques as quantum computing and neuromorphic architectures. Quantum computing exploits the physics of quantum mechanics to use the same circuits to perform many actions at the same time. Potentially some problems could be solved in seconds that would take conventional computers years to perform. Neuromorphic architecture is inspired by brain structures for such processes as pattern matching, searching, and machine learning. It is uncertain when such innovative concepts will achieve useful commercialization, but the future of computing systems and architecture is promising and exhibiting exciting potential.

1.6 THIS TEXTBOOK AS A GUIDE AND TOOL FOR THE STUDENT

This textbook is a graded introduction to the theory and practice of supercomputing. Each chapter brings in three or four key concepts, semantics, and technologies aimed at providing a *performance-oriented* and *systems-oriented* introduction to HPC. The topics are selected to provide both the theoretical background for understanding the abstract components of the field and a practical understanding of conventional practice needed to deploy applications, implement parallel algorithms, debug code, and monitor performance. While this textbook is intended for study under the guidance of an instructor in a college course, it is also suitable for individual study with the basic computing prerequisites covered in the appendices.

Chapters are organized to deliver three kinds of information: concepts, knowledge, and skills. Concept discussions aim to teach those ideas that have established theoretical foundations, enjoy longevity, and largely will not change. For instance, Amdahl's law, a performance model, is such a concept. Knowledge chapters aim to impart information about supercomputing to the reader that will evolve with time and will need to be added to in the future. For instance, the historical aspects of supercomputing fall into this category. Finally, skills needed for entry-level work in supercomputing are presented in tutorial style for ease of learning. These skills may change over time, but represent current conventional practice in the field. An example of this is the specifics on how to use the resource management tools on a supercomputer or how to program with a specific user application programming interface (e.g., OpenMP, MPI).

Exercises aimed at reinforcing understanding of the material are found at the end of each chapter. While much of the material will not require the use of a supercomputer to understand it fully, arranging access to a small cluster is recommended to attempt the practical coding exercises and examples given throughout the text.

Several of the chapters may serve as a reference guide to the specific technology they introduce. For example, the chapters on MPI, OpenMP, and essential resource management have also been written for self-contained reference usage.

The HPC field is replete both with freely available and proprietary software and toolkits designed to assist the practitioner and systems engineer in designing and deploying supercomputing applications. In general, this textbook provides a brief survey of such software and toolkits where relevant. However, for all examples and exercises only open-source and freely available software applications are utilized.

At the end of this course, the student can expect the following outcomes.

- A general overview of conventional practice in supercomputing in terms of both hardware architecture and software.
- A practical understanding of conventional practice in HPC software, including MPI, OpenMP, and OpenACC.
- A theoretical understanding of performance modeling and the key elements affecting parallel performance.
- A theoretical and practical understanding of file systems, resource management systems, debugging, and performance measurement.
- A theoretical and practical understanding of several key widely used parallel algorithms from a broad range of disciplines.
- A theoretical and practical understanding of the key operating systems in use by supercomputing systems.
- A broader view of the future directions of supercomputing in terms of both architecture and systems software.

1.7 SUMMARY AND OUTCOMES OF CHAPTER 1

- Definition of supercomputing and HPC. HPC incorporates all facets of three key disciplines: technology, methodology, and application. The principal defining property and value provided by HPC is delivered performance for an end-user application.
- Moore's law: the prediction by Intel cofounder Gordon Moore that device transistor density would increase by a factor of two every 2 years.
- Top 500 list. The Top 500 list ranks supercomputers in order of their performance running the HPL or "Linpack" benchmark for dense linear algebra. The list is updated twice a year.
- System stack of hardware and software comprising a supercomputer. The system stack of a supercomputer is a layered hierarchy of many physical and logical components, beginning with the system hardware and including processors, interconnection, and data storage. The system software that controls the hardware and manages the physical resources is associated with the

operating system, comprising both node-level instances controlling the node resources and the middleware which logically integrates many nodes and their local operating systems into a single system image. Software above the operating system abstraction includes resource management associated with executing user applications and workloads.
- Sustained and peak performance. Peak performance of a system is the maximum rate at which operations can be accomplished theoretically by the hardware resources of a supercomputer. Sustained performance is the actual or real performance achieved by a supercomputer system in the performance of an application program. While sustained performance cannot exceed peak performance, it can be much less.
- Benchmarks. The HPC community selects specific problems to compare and assess different HPC systems and capabilities. One of the most widely reported HPC benchmarks is HPL.
- Sources of performance degradation: starvation, latency, overhead, and contention. Starvation is when sufficient work is not available at any instance in time to support issuing instructions to all functional units every cycle. Latency is the time it takes for information to travel from one part of a system to another. Overhead is the amount of additional work beyond that which is actually required to perform the computation. Contention is when two or more requests are made at the same time and have to be serviced by the same single resource, either hardware or software, meaning that the requests can only proceed one at a time.
- Major epochs of supercomputing evolution based on technology drivers, execution models, and computer architecture. A perspective of seven epochs includes calculator mechanical technology, von Neumann architecture in vacuum tubes, ILP, vector processing, SIMD arrays, communicating sequential processes, and multicore petaflops.
- Possible future directions of HPC architecture. With the end of Moore's law, continued system performance growth will rely on other innovations through alternative device technologies, architectures, and even paradigms.

1.8 QUESTIONS AND PROBLEMS

1. Define or expand each of the following terms or acronyms.
   - HPC
   - Flops, gigaflops, teraflops, petaflops, exaflops
   - Benchmark
   - Parallel processing
   - OpenMP
   - MPI
   - Moore's law
   - Strong scaling
   - Starvation
   - Latency
   - Overhead
   - TLB, TLB miss
   - ALU
   - von Neumann architecture

  - Turing machine
  - SSI
  - DRAM
  - SIMD
  - VLSI
  - Distributed memory
  - Commodity cluster
  - NASA Beowulf Project
  - Communicating sequential processors.
2. What is the primary requirement that differentiates HPC from other computers? What other requirements are also important?
3. Describe four reasons for performance degradation using the acronym SLOW. Give examples of each.
4. Give six techniques noted in the text for improving performance.
5. Name and give a brief description of the seven epochs in the history of supercomputing.
6. Describe the computer recognized as the first true supercomputer. Who developed it, and what company did he later form?
7. Suppose you have a computer that has a four-stage pipeline and a workload with an input set of operand values of size 100. Assuming that each stage takes one unit of time and passing results from one stage to the next is instantaneous, what is the average parallelism in your computer for this workload?
8. Describe Beowulf computers, with at least five characteristics. What makes them significant to supercomputing?
9. Describe what is meant by the "end of Moore's law".
10. What was the fastest computer in the year that you were born? What technologies were used in that fastest computer? How much faster is the world's fastest computer today?

REFERENCES

[1] J.J. Dongarra, P. Luszczek, A. Petitet, The LINPACK benchmark: past, present and future (PDF), John Wiley & Sons, Ltd. Concurrency and Computation: Practice and Experience (2003) 803–820.

[2] T. Rauber, G. Runger, Parallel Programming for Multicore and Cluster Systems, Springer, 2013, ISBN 978-3-642-37800-3.

[3] The Potential Impact of High-End Capability Computing on Four Illustrative Fields of Science and Engineering, Committee on the Potential Impact of High-End Computing on Illustrative Fields of Science and Engineering and National Research Council, October 28, 2008, ISBN 0-309-12485-9, p. 9.

[4] J.P. Singh, D. Culler, Parallel Computer Architecture, Nachdr. ed., Morgan Kaufmann Publ., San Francisco, 1997, ISBN 1-55860-343-3, p. 15.

[5] J.L. Hennessy, D.A. Patterson, J.R. Larus, Computer Organization and Design: The Hardware/software Interface, second ed., third print. ed., Kaufmann, San Francisco, 1999, ISBN 1-55860-428-6.

[6] A.O. Allen, Computer Performance Analysis with Mathematica, Academic Press, 1994.

[7] B. Collier, J. MacLachlan, Charles Babbage: And the Engines of Perfection, Oxford University Press, September 28, 2000, ISBN 978-0-19-514287-7, p. 11.

[8] ENIAC in Action: What It Was and How It Worked, ENIAC: Celebrating Penn Engineering History, University of Pennsylvania. Retrieved 2017.
[9] M.V. Wilkes, Memoirs of a Computer Pioneer, MIT Press, Cambridge, Mass, 1985, ISBN 0-262-23122-0.
[10] M.D. Hill, N.P. Jouppi, G.S. Sohi (Eds.), Readings in Computer Architecture, Morgan Kaufmann, September 23, 1999, ISBN 978-1558605398, p. 11.
[11] The Cray-1 Computer System (PDF), Cray Research, Inc, 1978.
[12] C.J. Murray, The Supermen: Story of Seymour Cray and the Technical Wizards behind the Supercomputer, 1997, ISBN 0-471-04885-2.
[13] K.E. Batcher, Design of a massively parallel processor, IEEE Transactions on Computers. C 29 (9) (September 1, 1980) 836–840, http://dx.doi.org/10.1109/TC.1980.1675684.
[14] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, C.V. Packer, BEOWULF: a parallel workstation for scientific computation, Proceedings, International Conference on Parallel Processing 95 (1995).
[15] T.L. Sterling, Beowulf Cluster Computing with Linux, MIT Press, 2001, ISBN 0262692740.
[16] Blue Gene: A Vision for Protein Science Using a Petaflop Supercomputer (PDF), IBM Systems Journal, Special Issue on Deep Computing for the Life Sciences 40 (2) (2001).

CHAPTER

HPC ARCHITECTURE 1: SYSTEMS AND TECHNOLOGIES

2

CHAPTER OUTLINE

2.1 Introduction **44**
2.2 Key Properties of HPC Architecture **44**
2.2.1 Speed 45
2.2.2 Parallelism 45
2.2.3 Efficiency 46
2.2.4 Power 46
2.2.5 Reliability 47
2.2.6 Programmability 48
2.3 Parallel Architecture Families—Flynn's Taxonomy **48**
2.4 Enabling Technology **51**
2.4.1 Technology Epochs 51
2.4.2 Roles of Technologies 55
2.4.3 Digital Logic 55
2.4.4 Memory Technologies 58
2.4.4.1 Early Memory Devices 59
2.4.4.2 Modern Memory Technologies 61
2.5 von Neumann Sequential Processors **62**
2.6 Vector and Pipelining **64**
2.6.1 Pipeline Parallelism 65
2.6.2 Vector Processing 68
2.7 Single-Instruction, Multiple Data Array **69**
2.7.1 Single-Instruction, Multiple Data Architecture 69
2.7.2 Amdahl's Law 70
2.8 Multiprocessors **73**
2.8.1 Shared-Memory Multiprocessors 74
2.8.2 Massively Parallel Processors 76
2.8.3 Commodity Clusters 77
2.9 Heterogeneous Computer Structures **78**
2.10 Summary and Outcomes of Chapter 2 .... **78**
2.11 Questions and Problems **80**
References **82**

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00002-2

2.1 INTRODUCTION

High performance computer architecture determines how very fast computers are formed and function. High performance computing (HPC) architecture is not specifically about the lowest-level technologies and circuit design, but is heavily influenced by them and how they can be most effectively employed in supercomputers. HPC architecture is the organization and functionality of its constituent components and the logical instruction set architecture (ISA) it presents to computer programs that run on supercomputers. HPC architecture exploits its enabling technologies to minimize time to solution, maximize throughput of operation, and serve the class of computations associated with large, usually numeric-intensive, applications. In recent years supercomputers have been applied to data-intensive problems as well, popularly referred to as "big data" or "graph analytics". For either class of high-end applications, HPC architecture is created to overcome the principal sources of performance degradation, including starvation, latency, overheads, and delays due to contention. It must facilitate reliability and minimize energy consumption within the scope of performance and data requirements. Cost is also a factor, affecting market size and ultimate value to domain scientists and other user communities. Finally, architecture shares in combination with the many other layers of the total HPC system the need to make application programming by end users as easy as possible.

A number of classes of HPC architecture have been employed for different technological niches over the decades, each addressing these key performance issues in the context of their respective enabling technologies. A unifying theme across HPC architectures is "parallelism", meaning the ability to perform multiple actions simultaneously and thus reduce the total time to accomplish the combined tasks and operations of a user workload. The different classes of architecture introduced in this chapter reflect some of the most widely employed forms of parallelism. While discussion of HPC is often focused on the largest systems, the field spans a wide range of performance operating points. More important than specific design points is the ability to bring orders of magnitude more capability to a valued problem than would be possible with a conventional uniprocessor or personal workstation. Thus the effect that defines a supercomputer and differentiates it from commercial (or even consumer) servers is that it delivers greatly enhanced performance to solve real-world problems. This can be realized with even a modest parallel computer, far smaller than the number one machine on the Top 500 list, but still much faster than the machine on which this textbook has been prepared.

2.2 KEY PROPERTIES OF HPC ARCHITECTURE

HPC architecture extracts performance from the underlying enabling technologies for the range of applications deemed important in the context of the user institution's mission. The organization of the architecture incorporates structures of the components that make best use of the devices and the dataflow patterns that move information between them. Three key properties of an architecture determine delivered performance: the speed of the components comprising the system, the parallelism or number of components that can operate concurrently doing many things simultaneously, and the efficiency of use of those components in the degree of utilization achieved. A simple relationship of these key factors shows their contributions to delivered performance in Eq. (2.1).

$$P = e \times S \times a(R) \times \mu(E) \tag{2.1}$$

where P is average performance, S is scaling (the number of units that can operate at the same time), a (which is a function of reliability, R) is availability, which is the total fraction of time the system is capable of performing a computation, and μ is the instruction retirement rate (usually the clock rate) of the processor core, which is a function of the power, E. Average performance is normally reported in terms of clock rate, while S and a are generally reported without units.

2.2.1 SPEED

HPC system performance is directly related to the speed of its components. A key parameter is the clock rate of its constituent processor cores, or basically the rate at which each retires instructions. But the technologies employed for different functionality have widely differing speeds or cycle times. Much architecture is devising structures and methods that match these disparate speeds. For example, a major concern is the speed of the processor, again the clock rate, with the cycle time of the main memory. Processor core clock rates may vary from slightly less than 1 GHz to approaching 3 GHz, with a few more extreme examples in both directions. Memory cycle times presented by dynamic random access memory (DRAM) devices to the processors are in the order of 100 times longer (substantial variation depends on details). But there are other forms of memory, specifically static random access memory (SRAM) technology, that depending on size and power consumption can operate at or near the speeds of the processor core logic. A modern architecture will include a memory hierarchy consisting of a mix of slower higher-density DRAMs for capacity with faster low-density SRAMs called "caches" to achieve speed. A third aspect of speed is the rate at which data can be transferred or communicated between any two points within the system. Two measures are applied for this communication speed. The bandwidth determines how much information can be moved between two points in unit time or the rate of data movement. The latency measures how long it takes to move data between the two points. These too can vary dramatically depending on the distances between the source and destination, as well as the type of technology employed and the amount used. Architecture is, among other things, the art of balancing these different time constants in structures and through methods that will yield the overall best delivered performance for a user workload (e.g., application) within normalizing cost factors like nonrecurring engineering (NRE), deployment, power, and user productivity.

2.2.2 PARALLELISM

No matter how fast the speed of the parts technology can be, it will never be fast enough alone to deliver the necessary performance required by major application problems. Fundamental limits such as the speed of light, atomic granularity, and the Boltzmann constant constrain how fast a single processor core can execute a stream of instructions. Thus HPC architecture is heavily dependent on structures that permit many actions to occur simultaneously: the ability to do many things at once. This is referred to as "parallelism", and the many different classes of parallel computer architecture are defined and distinguished by the diversity of structures that are employed to achieve parallelism in different ways. But HPC architecture is also determined by how such parallelism is controlled. Thus both the data path and the control path are factors in how parallelism is exploited by an HPC architecture.

2.2.3 EFFICIENCY

The third factor that determines delivered performance for a user workload is efficiency. Efficiency is primarily the utilization of the system, or the percentage of time that the critical components are employed. This is more complicated than it suggests. The question is: upon which components should efficiency be measured? For HPC, a common measure of efficiency is the ratio of the sustained floating-point performance to the theoretical peak floating-point performance, both measure in flops, floating-point operations per second (please note that the "s" is not the plural, but rather stands for "second").

$$e_{flops} = \frac{P_{sustained}}{P_{peak}} \tag{2.2}$$

where e_{flops} is floating-point efficiency such that $0 \leq e_{flops} \leq 1$, P_{peak} is the theoretical peak performance of the HPC architecture measured in flops, and $P_{sustained}$ is the achieved average floating-point performance.

However, this typical measure of efficiency reflects an earlier era when a floating-point operation was expensive, either taking a long time to perform or requiring expensive and complicated floating-point hardware. Today data movement and the costs of data access from memory are far more significant in die space, time, and energy than a register-to-register floating-point operation. Nonetheless, this is the metric that is most likely encountered.

2.2.4 POWER

Every computer, large or small, uses electrical power for its operation. The speed of processor cores is in part proportional to their clock rate, and this in turn relates to the power applied. As long as a computer like your laptop does not consume more power than is available via the typical electrical infrastructure, this is not an issue. For example, a laptop computer will require perhaps 80 W or less sustained power. A deskside workstation may demand 200–400 W depending on such features as number of screens and amount of disk capacity. These are well within the capacity of the electrical service infrastructure of a light industry building, even for many worker stations. Electricity is not only required to deliver power to drive the many integrated circuits, interconnection channels, and input/output (I/O) devices but also to remove the resulting heat from the system.

Thermal control is essential if the system is not to overheat and ultimately fail as a consequence. A high-end processor socket may consume anywhere from about 80 W to more than 200 W. Small to modest-size computers are air-cooled. Cold air is forced through the system modules and over the processor, memory, and control sockets to remove the heat generated as they operate. For higher-wattage parts, substantial metal radiator hardware is fitted directly on to the sockets for thermal conductivity, providing greater surface area and cooling capacity. This reduces the density of packaging and ultimate computing capability per unit module. Additional electrical power is required to chill the air and force it through the systems. This can easily be as much as 20% of the total power budget. Liquid cooling exploits the higher specific heat of fluids, most often water, to increase packing density and enable higher-power parts for higher clock rates or larger logic dies. There is a wide range of liquid cooling systems and mechanisms, and hybrids of combined air and liquid cooling.

Active thermal control is becoming increasingly important and common among the new generation of high performance computers. Measurement of temperatures throughout the system, including key chip temperatures, allows monitoring of thermal gradients and can support thermal control. Modern multicore processors permit variable clock rates, voltage adjustment, and variable numbers of active cores, all of which facilitate achieving a balance between power and performance. This requires some level of software management, either to establish settings at the beginning of an application program execution or to adjust these settings continually during runtime as application demands change.

2.2.5 RELIABILITY

No systems operations are perfect, and the reliability of HPC systems is additionally compounded by their scale. Errors can occur periodically due to hardware or software faults. "Hard" faults occur when some part of the hardware breaks permanently, causing an intrachip component to fail, a core of a processor socket to be inoperable, or the entire socket to become useless. Hard faults can affect cores, memory, communications, secondary storage, and control. A "soft" fault occurs when a part intermittently fails but otherwise operates correctly. Such transient failures are due to a number of possible causes, including occasional cosmic rays or "noise" resulting from low voltage margins among others. Software errors are due to flawed coding of either the user application program or the supporting system software, such as the operating system. Programmer errors are routine, and a process of program debugging is a part of the task of application development. An interruption due to a mistake by the operating system is more difficult to deal with, as it is usually the province of the system vendor.

Different application problems may require different responses to lead to a final and correct solution. For large computations of big problems on high-scale machines, a common methodology is "checkpoint/restart". Periodically the system will stop a computation being performed and store all the program state at that point ("checkpoint"), usually on secondary storage. If an error occurs after that time in the execution, the problem need not be restarted from the beginning but rather from the last checkpoint. If the error is caused by a hard fault, the system has to be reconfigured to eliminate the broken hardware from the part of the system being used prior to restarting the application. If it is a soft error, the program can be restarted from the last checkpoint without reconfiguration. In the case of a software bug, the code will have to be corrected by the user or application supplier before it can proceed. In all three cases, diagnosis is required to establish the cause and possible source of the error before execution proceeds.

2.2.6 PROGRAMMABILITY

How difficult it is to write or develop a complex application code reflects the programmability of the system. While other parameters such as performance or power can be readily defined and quantified, programmability largely defies specification although there have been many attempts; for example, standard lines of code (SLOC). Although less straightforward to define, it is nonetheless extremely important to the overall utility of HPC. While the cost of deploying a major HPC platform may reach hundreds of millions of dollars, the cost of the software that runs on it may reach billions of dollars in total. Many factors contribute to programmability (or the lack thereof), including the processor core and system architectures, the programming models and facility of the language, the effectiveness of the system software such as compilers, runtime systems, and operating system, and the skills of the programmers themselves. The level of effort required to write the application is strongly related to the performance ultimately achieved. Within a range of behaviors, the greater the performance required, the harder it is to optimize the user program. This interrelationship between performance achieved and programming effort is sometimes referred to as "productivity".

Improving the ease of use of HPC systems for domain applications benefits from a number of techniques making up a discipline of code development. Indeed, the best way to write an application code is "do not". Many libraries of common codes have been developed by experts and optimized for a diversity of HPC system types and scales. Code reuse is critical to managing application development complexity and difficulty. An application program can become as simple as building a high-level framework that calls a sequence of existing library routines and passing data between them successively. When programs get very complicated and are borrowed by many different users, management of the code base itself can become challenging. The discipline of "software engineering" provides principles and practices that guide overall control of workflow management, including testing. These and other methods contribute to programmability.

2.3 PARALLEL ARCHITECTURE FAMILIES—FLYNN'S TAXONOMY

There are many distinct classes of parallel architectures. Further, individual architectures may be hybrids incorporating characteristics and strengths of more than one type. This section introduces parallel architecture families in terms of structure of concurrent processing components and their parallel control. This overview is intended to convey a sense of the alternatives available, their relationship to the underlying enabling technologies, and to some degree how they address the key challenges to achieving sustained performance.

The 1970s saw a rapid increase in the practical application of parallel architecture for supercomputing, with a number of technology and organizational choices available. Michael Flynn proposed a taxonomy that simplified categorization of distinct classes of parallel architecture and control methods based on the relationships of data and instruction (control) with respect to parallelism of "streams". Although of limited value today, this nomenclature has stuck and is, if nothing else, part of our culture and vernacular. It also established the notion of data parallelism and task parallelism, to be encountered later in greater specificity.

Comprising four characters, it divides the world of computing structures into four mutually exclusive, collectively exhaustive classes that can be viewed in a two-dimensional space. One dimension concerns the data stream, "D", and whether there is one such stream, "S", or multiple data streams, "M". The other dimension relates to the control or instruction stream, "I", and similarly whether here too there is only a single stream of control or multiple instruction streams. From these Flynn proposed a set of four four-character acronyms as a codification of the parallel architecture choice space. It is still used today, more than 4 decades later.

SISD—single instruction stream, single data stream (pronounced "sisdee"): this represents the conventional sequential (serial) processor structure where a single thread of control, the instruction stream, guides the sequence of operations performed on a single set of data, one operand at a time. In truth, this is even more simplistic than today's conventional uniprocessors, which actually have several operations "in flight" at any one time.

SIMD—single instruction stream, multiple data stream (pronounced "simdee"): the first form of parallelism conveyed within this taxonomy is simultaneous operation on multiple datasets, controlled by the same set of instructions. Thus each operation at any one time is the same performed on different data arguments. Although simple in concept, SIMD has had long-term impact, first as the basis for entire systems and later as part of more complex control structures in today's heterogeneous microprocessors and supercomputers.

MIMD—multiple instruction stream, multiple data stream (pronounced "mimdee"): this category suggests that, like SIMD, there are many sets of data but in this case each dataset has its own instruction stream associated with it. At any one time there are many operations being performed, but they need not be the same and in fact are almost always different. As will be seen, this is the most widely used form of parallel architecture, but the category has many different subclasses.

MISD—multiple instruction stream, single data stream (pronounced "misdee"): surprisingly, the fourth of Flynn's categories is controversial, with some practitioners of the field considering it meaningless. It is not. One possible interpretation is a coarse-grained pipeline where each pipe stage accepts data from the previous stage, performs a set of operations on these data stream elements, and then passes on the results to the next stage. Another interpretation is a shared-memory multiprocessor (Section 2.8.1) where, as the name suggests, multiple processors each with its own instruction stream work on the same (therefore shared) data on which all the other processors operate.

One last related term, SPMD (pronounced "spimdee"), while not strictly part of Flynn's taxonomy, is related to and inspired by it. SPMD stands for "single program, multiple data stream" and reflects a practical variation of the SIMD model. Instead of issuing and broadcasting one instruction at a time to all the simple processing units of a SIMD-like machine, SPMD sends a function call of a coarse-grained procedure that is to be performed by all the processing units of the parallel machine. The invocation of heavyweight tasks rather than lightweight instructions amortizes the overheads and latency times involved in system control, and enables the operation of some forms of modern computing structures, including graphics processing unit (GPU) accelerators (Fig. 2.1) (Chapter 15).

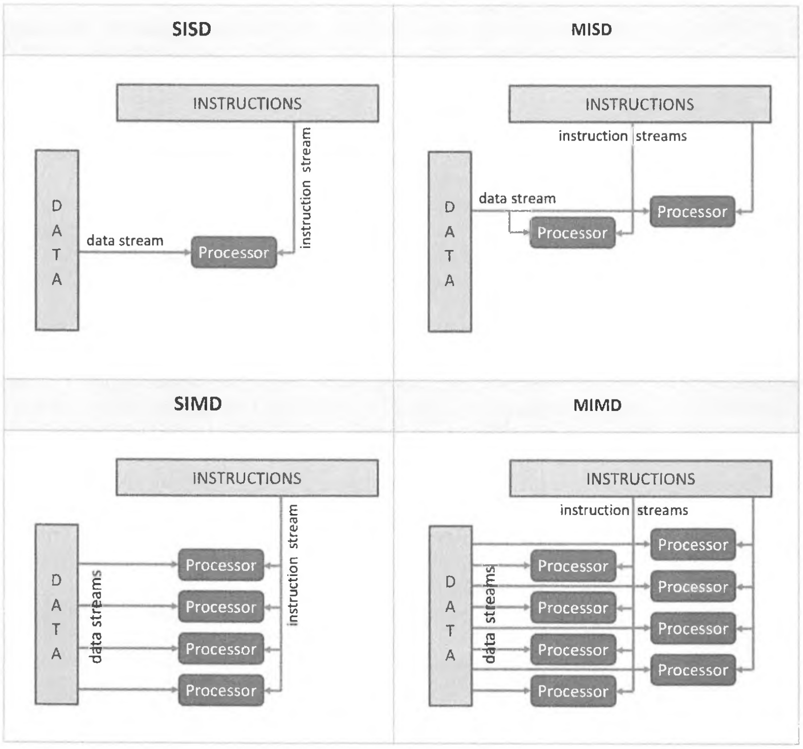


FIGURE 2.1

Flynn's taxonomy. *MIMD*, multiple instruction stream, multiple data stream; *MISD*, multiple instruction stream, single data stream; *SIMD*, single instruction stream, multiple data stream; *SISD*, single instruction stream, single data stream.

2.4 ENABLING TECHNOLOGY

HPC systems are products of opportunity enabled by device technologies. Such technologies may be derived through development unrelated but useful to HPC, created for the purpose of advancing computing, or result from incremental enhancements or extensions to existing HPC component types. Early digital electronics technologies were derived from radio and radar devices. Magnetic core memory was created to revolutionize data storage, which it did. Very large-scale integration (VLSI) circuits spanned more than 2 decades of continued improvement through increases in semiconductor device density and switching times. HPC architecture is strongly influenced by existing and emerging technologies to make best use of the opportunities that they may deliver and adjust to the challenges they impose.

2.4.1 TECHNOLOGY EPOCHS

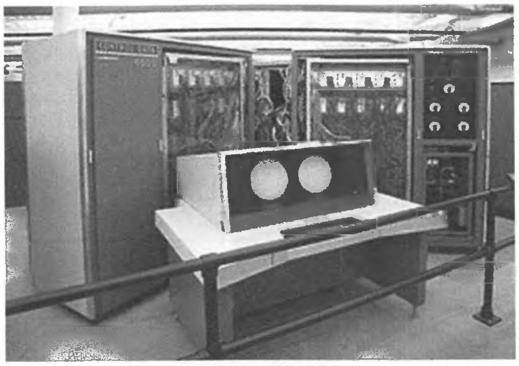
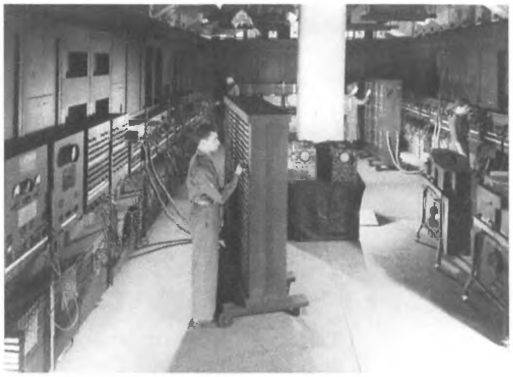


Photo by Jitze Couperus via Wikimedia Commons

The Control Data Corporation (CDC) 6600 computer released in 1963 could perform over 3 million operations per second, operated at 10 MHz, was roughly 50 times faster than its immediate predecessor, the CDC-1604 released in 1960 (the fastest machine in the world at the time of its release), and was three times faster than the fastest IBM machine at the time, the IBM 7030 Stretch. The CDC-6600 was designed by Seymour Cray with a remarkably small staff of 34 people (with only one PhD among them) at his Chippewa Falls, Wisconsin, laboratory, and was innovative in using silicon-based transistors from Fairchild Semiconductor (cofounded by Robert Noyce, future cofounder of Intel Corporation) and Freon cooling rather than air cooling. Because of the unique cooling arrangement, the CDC-6600 was physically smaller than its CDC-1604 predecessor even though it was 50 times faster. The CDC-6600 was an enormous commercial success, selling over 100 units each costing US$8 million. Many of these machines were used at US national laboratories as part of the nuclear weapon simulations which helped contribute to the negotiations of the nuclear test ban treaty with the USSR in 1963.



US Army photo of the ENIAC via Wikimedia Commons

The electronic numerical integrator and computer (ENIAC) was the first large-scale electronic digital computer that could be reprogrammed for running different applications. It was developed at the University of Pennsylvania under contract by the US Army, and consisted of 18,000 vacuum tubes requiring 150 kW to operate. Its genesis was in World War II, when requests for improved tables for artillery and bombing overwhelmed the personnel developing such tables using mechanical calculators. Interestingly, at that time those people were called "computers". Unlike previous electromechanical devices, the ENIAC internals contained no mechanical moving parts, enabling it to produce an entirely electronic computation and significantly speed up work. It was reprogrammed by a lengthy process of manually changing switches and cables; even so, it was still used for over 9 years from 1946 to 1955. In April and May 1948, the first-ever electronic Monte Carlo simulations were successfully performed on the ENIAC, simulating the diffusion of neutrons. Monte Carlo methods remain today a mainstay of scientific computation.

Historically, epochs of device technology spanning the centuries of calculating machines may be delineated by key transitions of the components of which they are comprised. The following dates are approximate, with substantial overlapping between successive periods of technology adoption.

- 3000 BCE—primitive counting devices: enumeration or counting of amounts of important stock like domesticated animals, agricultural produce (units of grains, containers of olive oil), and products like lengths of textiles even at early stages of civilization in the Bronze Age and before required mechanical means of recording (storing) quantities and performing simple additions and subtractions to abstract actual real-world items. This inaugurated data storage, calculation, and abstraction through mechanical means. The abacus used in some areas up until the postwar era is a direct derivative.
- In 200—100 BCE one of the earliest known analog computers was developed, consisting of more than 80 pieces and 30 mechanical gears. Known as the Antikythera, this instrument was used for astronomical predictions (Fig. 2.2). The technology for this device was subsequently lost, and any widespread cultural impact of the Antikythera mechanism on the ancient world is generally considered controversial.

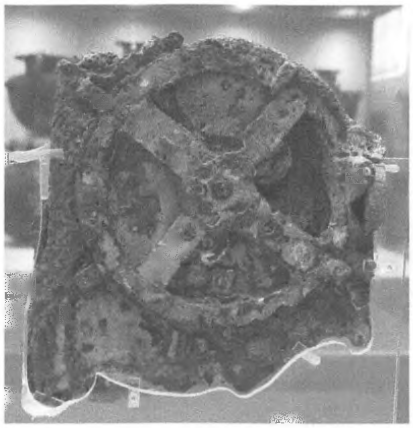


FIGURE 2.2

The Antikythera mechanism, one of the earliest known analog computers.

By Tilemahos Efthimiadis via Wikimedia Commons

- 1600—mechanical devices with gears and levers: the evolution of clockwork mechanisms, which were applied to relatively compact, reliable, and eventually sophisticated calculating devices that mastered sequencing of microoperations such as carries in decimal addition and multiplication. The Pascaline (see Fig. 2.3) developed by the mathematician Pascal is one of many such calculating devices, culminating in significant calculating engines like the difference engine invented by mathematician Charles Babbage in the early 19th century.
- 1850—electromechanical: motors, relays, punched cards. The inauguration of electricity offered new media for basic operation, sequencing, and data storage, both temporary and persistent. The tabulator (Fig. 2.4) devised by Hollerith for the 1890 US census led to the founding of IBM and 50 years of commercial data processing. The Mark I developed by Aiken at Harvard and

FIGURE 2.3

The Pascaline mechanical calculator.

By David Monniaux via Wikimedia Commons

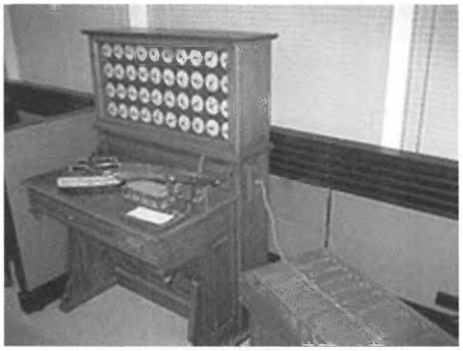


FIGURE 2.4

The tabulator built for the 1890 US census.

Photo by Adam Schuster via Wikimedia Commons

manufactured by IBM was a culmination of the complex calculating systems enabled by these technologies, delivering approximately 1 operation per second (ops). Processor cores that separate instruction streams from data streams are still known as "Harvard architectures" today.

- 1940—vacuum tube: logic gate, flip-flop, magnetic core. With the emergence of electronics, initially by means of the amplifying vacuum tube, and their incorporation in digital logic by Atanasoff in the 1940s, calculators such as the US ENIAC and British Colossus increased computing rates 1000-fold and inspired the advanced architecture concept attributed to the mathematician John von Neumann, providing the foundation of the digital programmable computing paradigm. The three pivotal elements of the modern computer were firmly established and integrated into a form of which most future models would be derivatives. These elements were memory implemented as magnetic cores (old terminology; not processors), digital electronic logic using Boolean and binary encoding, and communication via digital signals through electrical wires.
- 1955—transistor: the replacement of vacuum-tube technology with semiconductor technology (germanium and silicon) dramatically reduced power consumption, cost, and size while greatly increasing speed and reliability.
- 1965—integrated circuits (small-scale integration/medium-scale integration): the placement and interconnection of multiple transistors with other components (e.g., resistors and capacitors) heralded another stage of cost and power reduction with speed and reliability increase by modularized logic gates on a single semiconductor die or "chip". The concept of the architecture family was introduced with the IBM 360 ("mainframe") and the Digital Equipment Corporation (DEC) PDP-8 ("minicomputer"), where multiple versions of computers with the same logical ISA could be sold with different performance-to-cost market points. Intermediate binary values were stored temporarily in semiconductor latches, with all intercommunication between functional modules encoded as digital signals. The CDC-6800 using multiple processing units and parallel ISA was among the first computers to deliver one megaflops.
- 1975—large-scale integration (LSI): large arrays of gates on a single chip permitted increasingly complex digital functional units to be implemented on single semiconductor dies, with core memory being replaced by semiconductor DRAM over this period. "Bit-slice" components allowed

full computers to be implemented with relatively few parts, yet permitted a wide diversity of ISAs through microcoding. High-speed technologies, although of lower density, drove new classes of supercomputers, like the Cray-1 vector processor delivering peak performance in excess of 100 megaflops. During this era, LSI also enabled the modern SIMD array computers such as the Maspar-1 and the CM-2. The first microprocessors were commercialized with 4-, 8-, and 16-bit architectures, and, while limited in performance, they dramatically reduced the costs of commercial and early consumer-grade computer electronics, including the first video games.

- 1990—VLSI with complementary metal-oxide semiconductor (CMOS): with increased device density through VLSI, significant single-chip microprocessors were possible, ultimately leading to 32-bit and eventually 64-bit data path architectures. These sparked a revolution, with the "killer micro" replacing more discrete component processor designs and ensembles of microprocessors replacing other forms of supercomputers. While a diversity of such multiprocessors were developed, three general classes emerged as dominant: the symmetric multiprocessor (SMP), the massively parallel processor (MPP), and the commodity cluster (e.g., Beowulf). Each has a different performance-to-cost design point.
- 2005—multicore heterogeneous with GPU: the modern era of technology and architectures emerged with the two combined trends of stagnant processor speed and multicore chips. Due to power constraints clock rates have remained relatively constant, although there is a significant spread of their values from below 1 GHz for embedded and mobile processors to over 3 GHz for the fastest. The continuing progress of Moore's law has made possible the incorporation of multiple processor cores on a single die to increase aggregate performance per chip. Special configurations of processing elements (PEs) can greatly accelerate important functions. The current era of supercomputers employ these two strategies to address these trends.

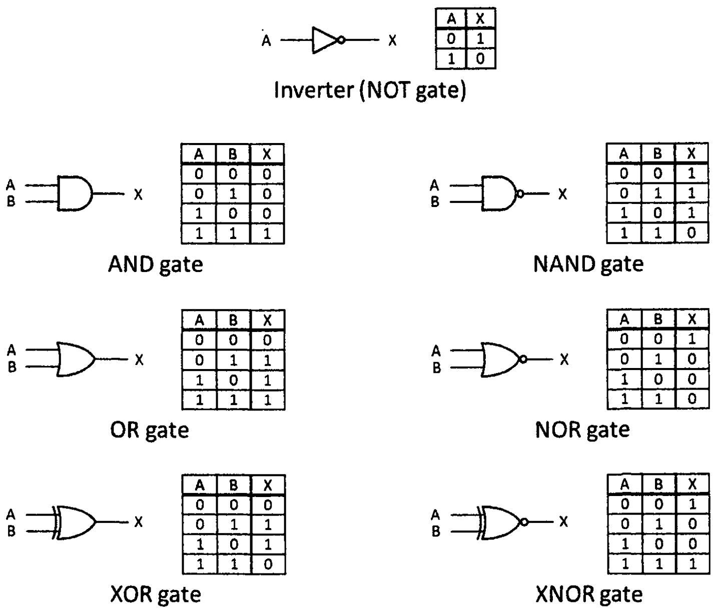
2.4.2 ROLES OF TECHNOLOGIES

Technologies play many different roles in enabling the implementation of computing systems and supporting their functionality. Three dominant classes of technologies largely define the design space for how HPC architectures evolve and the performance they are able to achieve. Throughout this text these fundamental aspects of HPC system implementation and operation will be examined. Here the dominant functional roles are introduced. The first technology class, digital logic, makes possible the actual basic operations that perform the calculations comprising the end-user work. It transforms one or more sets of input values, usually encoded as binary or Boolean information, into output values determined by the specified function (e.g., integer addition). The second technology class, memory, allows information to be stored temporarily (ephemeral) or permanently (persistent), to be accessed by logic elements, and to be modified (updated) as required. Memory technologies, even in a single architecture, are of a number of types that vary in speed and capacity. The third technology class, data communication, moves information from one part of the system architecture to another.

2.4.3 DIGITAL LOGIC

Digital logic technology is the workhorse of computer architecture. It occupies roles in every part of the computer system, from performing the actual operations of a calculation to controlling the memory subsystems and data communications. Digital logic is hierarchical (like architecture itself), in that the simplest devices are organized to create basic Boolean gates, which in turn are used to make more sophisticated functional units, and so on. The basic technology is an on/off switching device that permits or impedes the flow of electric current based on the state of an input signal. Alternatively, such switching devices determine the voltage exhibited at an output device, usually either a 0 V level or an alternative nonzero level to distinguish between two distinct associated Boolean or logical values, false or true, respectively, or "0" or "1" as they are often depicted. Over the last 7 decades these most basic switching devices have been successively vacuum tubes, discrete transistors, and integrated transistors (multiple transistors on a single semiconductor die).

Depending on the exact circuit design and basic physical technology, a number of switching devices are structured to work as logical gates or other simple functional units accepting one or a few input values (equivalent to Boolean 1 and 0) and producing one or more output values. The basic two-input logic gates represent every possible logical outcome, of which there are 16, including fixed values, invert, and, nand, or, nor, xor, and others, as shown in Fig. 2.5. Although circuits differ, typical gates are implemented with a dozen or so transistors. Key metrics of logic gate operation include switching rate and propagation delay. Switching rate is the highest frequency at which a logic gate output can change from a logical 1 to 0 (or 0 to 1) and back again. It is usually measured in gigaHertz (GHz) or billions of cycles per second. Propagation delay is the amount of time required for



A	X
0	1
1	0

AND gate

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

NAND gate

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

OR gate

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

NOR gate

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

XOR gate

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

XNOR gate

A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

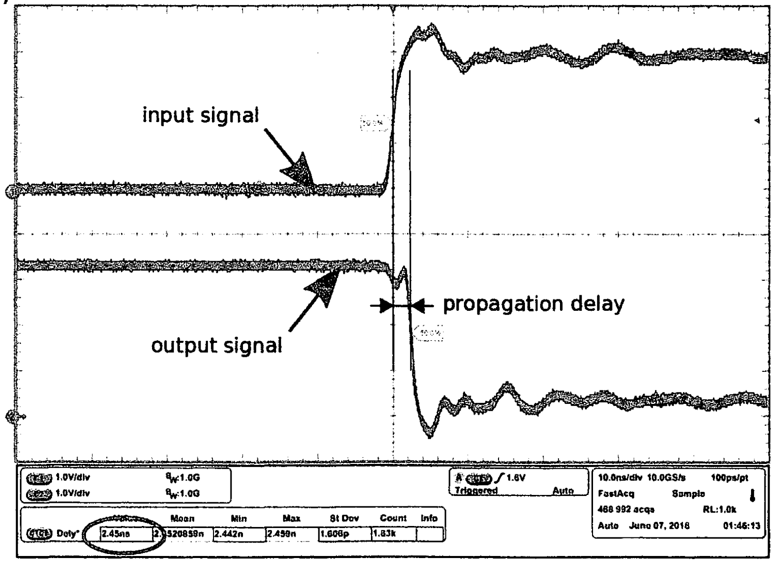
FIGURE 2.5

Basic two-input logic gates with corresponding logical functions.

a change in value of an input of a gate to be reflected by a corresponding change on the gate's output. This is usually measured in picoseconds or trillionths of a second. Fig. 2.6 illustrates these measures for an SN74AHC04 CMOS inverter manufactured by Texas Instruments, collected using a Tektronix MS5104 oscilloscope. A relevant excerpt from the datasheet for this integrated circuit is also shown in Fig. 2.6. Since propagation delay typically depends on the output load, such as the number of other gates driven by it, the workbench result differs somewhat from the specification.

Due to the limited number of states assumed by each connection in digital logic, representation of more complex concepts requires collections of multiple binary lines. To express integer numbers, an

(A)


input signal
output signal
propagation delay
1.0V/div
1.0V/div
B_W:1.0G
B_W:1.0G
1.6V
Triggered
Auto
10.0ns/div 10.0GS/s 100ps/pt
FastAcq Sample
468 992 acqs RL:1.0k
Auto June 07, 2016 01:46:13
Mean Min Max St Dev Count Info
Dely* 2.45ns 2.4520859n 2.442n 2.459n 1.606p 1.83k

(B) SWITCHING CHARACTERISTICS

over recommended operating free-air temperature range, V_{CC} = 3.3 V ± 0.3 V (unless otherwise noted) (see Fig. 2.1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	LOAD CAPACITANCE	T_A = 25°C		T_A = –55°C TO 125°C		T_A = –40°C TO 85°C		T_A = –40°C TO 125°C Recommended		UNIT
						SN54AHC04		SN74AHC04		SN74AHC04		
				TYP	MAX	MIN	MAX	MIN	MAX	MIN	MAX	
t_{PLH}	A	Y	C_L = 15 pF	5[1]	8.9[1]	1[1]	10.5[1]	1	10.5	1	10.5	ns
t_{PHL}				5[1]	8.9[1]	1[1]	10.5[1]	1	10.5	1	10.5	
t_{PLH}	A	Y	C_L = 50 pF	7.5	11.4	1	13	1	13	1	13	ns
t_{PHL}				7.5	11.4	1	13	1	13	1	13	

FIGURE 2.6

Timing in digital logic: (A) annotated oscilloscope trace showing propagation delay for a single inverter and (B) datasheet specification for the measured circuit.

Image (B): Courtesy Texas Instruments

ordering is imposed to signify the power of two each line's position is associated with, very much like the position of every digit in commonly used decimal numbers is associated with units, tens, hundreds, and so on (consecutive powers of 10). By convention, the least significant digit is written at the rightmost position in the number. For example, to express decimal number 6 as a binary number, it has to be converted to the sum of powers of two: $6 = 4 + 2 = 2^2 + 2^1 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 110_2$ (subscript 2 denotes the radix, or the base of the number, to avoid confusion when numbers in different bases are used in the same place). The reverse conversion requires summation of powers of two corresponding to the positions of ones in a binary number: $1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$. Binary numbers consist typically of significantly more digits, or *bits*, than the equivalent decimal numbers. To keep verbosity in check, octal (radix 8) or hexadecimal (radix 16) notation is frequently used as an alternative to a binary base, while offering ease of conversion to and from the actual binary format. Each octal digit, spanning values from 0 to 7, represents an arbitrary group of three binary digits, while a hexadecimal digit replaces four bits. Since there are no decimal digits to express values from 10 to 15, letters A through F (upper or lower case) are customarily used to express them. This is illustrated in Fig. 2.7.

2.4.4 MEMORY TECHNOLOGIES

Memory technology in its alternative forms enables the storing, access, and changing of data. As in the case of digital logic, information is represented by collections of bits. Each bit is 0 or 1 (alternatively true or false), and they are usually grouped as 8-bit bytes or multiple byte words. Information is treated and encoded as distinct types, such as Boolean, character, strings, integers (of different lengths), and floating point (32 bit or 64 bit), among others.

(A)

Octal digit	Binary equivalent
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

(B)

Hexadecimal digit	Binary equivalent	Hexadecimal digit	Binary equivalent
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

FIGURE 2.7

Conversion between (A) octal and (B) hexadecimal and binary bases.

2.4.4.1 Early Memory Devices

WHIRLWIND—THE FIRST SUPERCOMPUTER

The Whirlwind vacuum-tube-based digital electronic computer was the first modern computer architecture and represented the state of the art in high-speed calculation. It may be considered the first general-purpose supercomputer. Developed at Massachusetts Institute of Technology (MIT) under successive projects sponsored by the US Navy and then the US Air Force, its intended applications stressed performance initially for flight simulation and ultimately for radar-based air defense. Whirlwind employed a bit parallel logic design with 16-bit words performed and simultaneously implemented with vacuum tubes. It stored and controlled access to 2048 words using electrostatic storage tubes with an original (never achieved) bit density of 1024 bits per unit. The control structure incorporated an innovative diode matrix for speed as well as simplicity and flexibility of design. Its initial design was completed in 1947 by Jay Forrester and Robert Everett; it became operational in 1951 and consisted of 5000 vacuum tubes. Whirlwind was upgraded in 1953 with a new kind of memory developed by Forrester that used arrays of magnetic cores in stacks, replacing the slow and less reliable vacuum-tube storage. The resulting performance of up to 40 K instructions per second made Whirlwind the fastest computer of its time, dramatically increased its reliability and reducing its cost of operation.

The Whirlwind computer and its many innovations had far-reaching impacts on the field of computing. The invention of core memory redefined computer architecture for the next 2 decades, and is one of the main reasons why digital computers became commercially practical. Bit-parallel logic units became the norm for data processing. The diode—matrix control unit inspired Maurice Wilkes to conceive of microcontrollers and microprogramming, upon which future computers would be based at least until the microprocessor era. Whirlwind was the prototype for the first major parallel computer system, SAGE, employed as the original US air defense system. A spinoff of the Whirlwind project was the founding of DEC, which invented the minicomputer and rose to become the world's second-largest computer company in the 1980s. A second spinoff, MITRE, a major defense research contractor, can also be attributed to Whirlwind. With the final operational deployment of the Whirlwind computer, the future direction of high performance computer architecture was established.

History's earliest forms of memory predate the use of the abstraction of the bit and took such primitive forms as grooves on wooden sticks, marks on clay and stone tablets, pebbles used for counting (sometimes in depressions of wooden boards or tables), and beads on rows of horizontal rods (of wood or metal). In the age of enlightenment starting in the 17th century and extending well into the 20th century the position of gears served as storage, often distinguishing among 10 items in support of the decimal system.

With the emergence of digital electronics enabled by the vacuum tube and derived from analog electronics components such as radio, amplifiers, and radar, a clear need was seen for a technology that could store information to complement the logic that was performing operations on that information.

The first generation (1940–1952) saw many memory technologies devised and applied as part of the earliest digital computers, which were the supercomputers of their day (compared to rooms full of women with mechanical calculators). Among the earliest were mercury delay lines, also called "tanks", developed for radar uses during World War II. A tank was, as the name implies, a container, filled with the liquid metal mercury. At one end of the tank was an acoustic speaker that would create sound impulses into the medium. At the other end a microphone was positioned to detect the same sound signals. A closed loop was created in which the detected sounds from the microphone (acoustic sensor) were amplified (electronically) and fed back to the speaker at the top of the tank. Thus any individual bit of information would continue in a perpetual loop. The number of bits that could be stored in the mercury tank was a function of the maximum pulse repetition frequency, the length of the tank, and the propagation delay of the medium. This is one form of a dynamic memory in which the information has to be continuously refreshed. It was also referred to as a "one-dimensional memory" because of the serial nature of the bit availability. Mercury delay lines were used in such first-generation computers as EDSAC [1] and the IBM 704 [2].

A two-dimensional memory was also developed in the mid-1940s: it stored electrostatic charges on a phosphorous screen on the inside of a vacuum tube, very much like the old-fashioned video tubes in televisions and oscilloscopes. Originally named the Williams tube after its British inventor, the small regions of charge were created on the rectangular surface and could be viewed from the outside as an array of glowing bits. This vacuum-tube memory was also dynamic, as the stored charge would slowly "leak" off of the phosphorous surface and have to be rewritten every few milliseconds. The capacity of the memory was dependent on the surface area of the screen and the granularity of the charge bit cells. The speed of the memory was a function of the electronic delays to send a signal to hit the bit cell with an electron beam and detect its scattered charge (if one was there). It was more than an order of magnitude faster than the mercury delay line memory. Vacuum-tube storage was used in such computers as EDVAC [3] and IAS [4].

The breakthrough storage technology of the first generation of digital electronic computers was "core memory". Developed as part of the MIT Whirlwind [5] project in the late 1940s and manufactured by IBM, core memories used doughnut-shaped ferromagnetic beads to represent bits and exploited the hysteresis of the magnetic properties to store statically the equivalents of 1s and 0s. If there was a magnetic flux in one direction around the core, it would be a 1; if there was no flux in the core, it represented a 0. Core memory was organized in stacks of planes of such cores, referred to as three-dimensional memories. Three wires went through the center of a core being accessed. Electric currents through two of the wires were applied at the same time to provide enough stimulus to cause a core bit to set its flux. When reading the memory, the third wire is used to sense the change in flux (or absence thereof). While static, in that it can retain state indefinitely without active recharge, it is an example of destructive reading. When a bit is read, the state is potentially erased in the process and has to be reset. Nonetheless, core memory enabled the modern digital electronic computer. It was faster than any other storage technology, higher density, lower cost per bit, and lower energy. Its impact was so extreme that it was employed for more than 2 decades and almost immediately replaced the earlier storage technologies described above.

Magnetism as a physical phenomenon has played a major role in data storage, far beyond its application to core memory. As the key element of a thin veneer, ferromagnetic oxides have been applied to strings, tapes, drums, hard disks, and floppy disks. In this general form, it represents among the longest-lasting technologies in computing, used from the 1940s to the present mass-storage systems. Its principal value has been the combination of density, cost, and persistence. Some machines, such as the IBM 1620 [6], used it as a main memory due to the relatively low cost. Today all mass-storage systems comprise giant arrays of hard drives, possibly in combination with tape-drive robots for even greater capacity.

Lest we forget, paper (yes, paper) has a long tradition in data storage and was among the longest and most effective technologies employed for the purpose. While usually overlooked, paper served directly as "scratch pads" for people to write partial results of calculations for many hundreds of years. More directly, throughout the history of modern computing printers employ paper as the primary medium of storing and presenting output results to end users. Mechanical printer devices dominated this last stage of computing over the first 40 years of its history, although the work is now primarily performed by laser printers (and some ink-jet printers). Punched cards and later paper tape used paper products with punched holes to represent data. Punched cards were first devised for Jacquard's loom to control the patterns for weaving cloth, introduced in 1801. Cards were connected in a sequence as a chain with the holes in the card controlling the threads that were woven cross-wise to make the fabric. Later Hollerith developed the tabulating machine for the 1890 US census. Each card represented a US citizen, and the holes encoded various characteristics. This was the supercomputer of the day and led to the establishment of IBM, with punched cards among its core technologies and employed well into the 1960s. Punched cards were used not just for data but to represent lines of code to describe computer programs, one line per card. The pattern of punched holes to represent characters was referred to as "Hollerith code". Punched cards were also used as a medium of output as well as data input and source code. Card punches were connected as output devices to early computers (e.g., the IBM 360) for the results of user programs to be returned. Paper tape in rolls or fan-folded provided a cheap way of storing program code and data for low-cost computers like the minicomputers of the 1960s and 1970s.

2.4.4.2 Modern Memory Technologies

Modern computer architecture incorporate three principal memory technologies dominant in supercomputing: DRAM, SRAM, and magnetic storage media, including hard-disk drives and tapes. A fourth, nonvolatile random access memory (NVRAM), is emerging as a technology sitting between DRAM and mass storage.

SRAM provides the highest-speed semiconductor memory. But it is also the largest, taking up the greatest die area, and it consumes by far the greatest power. SRAM cells use a number of transistors—between 6 and 12, depending on how they are employed within the total logic structure (Fig. 2.8A). The fastest SRAM devices are used for processor core registers and latches, and can operate at processor clock rates of less than 1 ns. These are relatively low density. As is discussed in future chapters, small memory blocks referred to as "caches" are used to take advantage of locality to give the effect of fast memory while holding only a small portion of the total program data. Caches may be divided into levels, with the L1 cache the fastest but the smallest, providing a throughout of one word per clock cycle. L2 caches can be much larger than L1 and hold much more data, but operate slower in the order of 4—20 cycles. A third level, the L3 cache, may also be included, but these are not usually SRAM.

The main memory is the primary component for storing data within a computer and is composed almost entirely of DRAM technology. DRAM is much denser than SRAM, holding far more bits per unit area. It consumes much less power as well, but it is much slower. It can take between 100 and 200 cycles for a processor to access data from DRAM. Each bit cell of a DRAM chip consists of a single transistor and a capacitor (Fig. 2.8B). A capacitor can store an electrostatic charge difference which represents a Boolean or binary state value of "1". No charge difference represents "0". The single

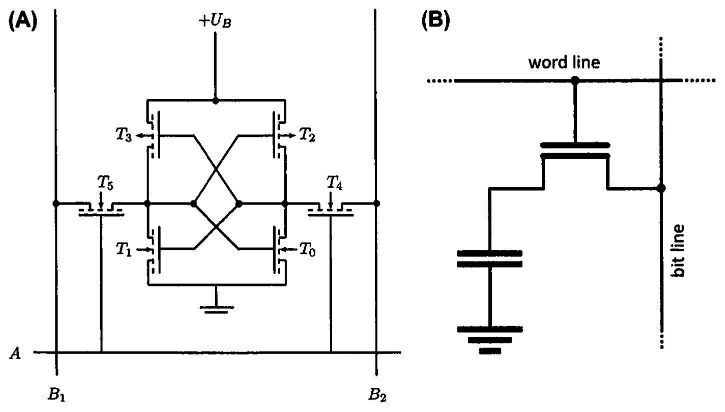


FIGURE 2.8

Single-bit storage cell structure in (A) static random access memory and (B) dynamic random access memory.

Image (A) by Martin Thoma via Wikimedia Commons

transistor isolates the capacitive charge from the sense lines to hold it and retain its state. Unfortunately, DRAMs are volatile. They require a destructive read where accessing a DRAM cell destroys its value and it has to be rewritten. Also, leakage from DRAM cells requires that they be refreshed in the order of tens of milliseconds.

2.5 VON NEUMANN SEQUENTIAL PROCESSORS

Although the word "supercomputer" may invoke mental images of specialized Cray vector machines or large concurrent-array systems, in fact the very first electronic digital stored-program computers were the supercomputers of their day. They delivered a level of performance 1000 times that of previous methods of calculation. The von Neumann architecture of sequential processing represents an important starting point in understanding how HPC is achieved. Initially a supercomputer in its own right, the original von Neumann architecture concepts and elements permeate in one form or another most of the modern and certainly the dominant form of supercomputer execution strategies.

Fig. 2.9 represents the principal elements of the von Neumann architecture conceived by Eckert, Mauchly, and the mathematician John von Neumann in the mid-1940s that has provided the recipe for most computing over the last 7 decades, admittedly with dramatic enhancements. This simplistic diagram offers an idealized picture of a sequential architecture. Nonetheless, most of the complicated elaborations have been devised to retain the image of this more perfect form and function. And while many of the key factors influencing performance are not shown by this template, speed is there, and provides a starting point to consider the extensions and elaborations embodied by the processing systems of today.

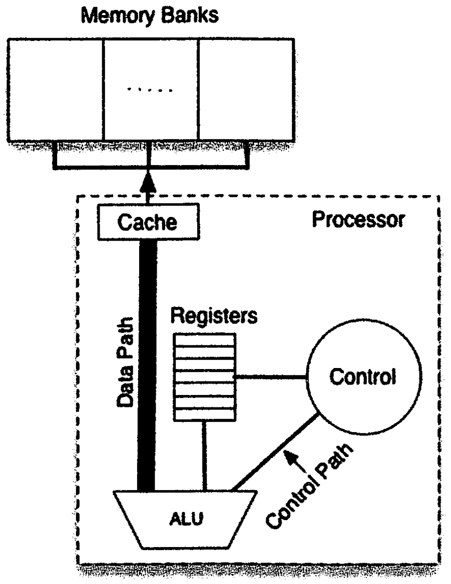


FIGURE 2.9

The principal elements of the von Neumann architecture. The caches and register banks were introduced at a later date.

Historically, the arithmetic logic unit (ALU) was considered the heart of the computer. It performed the actual numeric, character, and logical (Boolean) operations of the computation. It operates on argument values presented to it from some form of high-speed buffers, latches, or registers (shown here). Registers can be read and written to at the speed of the surrounding logic. In the earliest architectures, a single register, the accumulator, was used with an instruction set that referenced it implicitly. A bank of several registers has multiple ports so that simultaneous reads and writes can be performed. Unlike addresses for memory, registers have their own namespace, which usually cannot be operated on but only referenced.

The processor accesses the main memory of the computer system to store and use the values of the program variables making up the state of the calculation. While at first available main memory data was measured in thousands of bytes, today a typical processor core has direct access to billions of bytes (gigabytes) of memory. The largest HPC systems have an aggregate memory capacity in the order of a petabyte of main memory. A main memory is referred to as random access memory, as any of the storage locations may be designated by its hardware address to select one byte or word from the many. A load operation by the processor reads a word from the main memory and places its value into a designated register, to be used by the ALU at a later time. Because the cycle time of the memory is about two orders of magnitude slower than the clock rate of the processor, an intermediate level of storage referred to as "cache" is incorporated in the processor. Data that is accessed from memory is also copied to cache, with the expectation that it may be accessed again (a pattern referred to as temporal locality). When accessed in the future, the cached data can be acquired much faster from the cache than from the main memory itself. But the cache is much smaller than the main memory, so only some of the data can reside in the cache at any one time. Finding a sought-for value in cache is referred to as a "cache hit". The converse, not finding a required variable value in cache, is appropriately designated a "cache miss".

The operation of the processor is managed by the controller, which creates a sequence of signals to the hardware. Originally this was done as a series of phases: fetch instructions, execute operation, and write back to register (or memory). This would be repeated for each succeeding operation of the instruction stream. Far more complicated control sequences are required today, with multi-operation instructions performed with out-of-order completion, speculative execution for conditionals, reservation stations, and other advances to be considered in future chapters, all to achieve superior efficiency and ultimately performance.

2.6 VECTOR AND PIPELINING

EARTH SIMULATOR

Photo by Manatee_tw via Wikimedia Commons

The Earth Simulator (ES) was a hallmark supercomputer developed and deployed by the Japanese, with operations beginning in 2002 at the Earth Simulator Center in Yokohama, Japan. As measured by the highly parallel Linpack (HPL) benchmark, it was the fastest computer in the world then and for two more years, with a delivered performance of 35.9 teraflops. The ES is a milestone in supercomputing, as it marks the halfway point logarithmically between the first documented systems of the Top 500 list in 1993 and the current highest-rated system, which spans a range of approximately six orders of magnitude performance gain. The ES was a game changer, being about five times as fast as the previous top machine.

The ES was architecturally an elegant machine, built by NEC based on its SX-6 vector processor. It was an MPP with 640 nodes, each with eight vector processors that operated at a 3.2 GHz clock rate. Its total memory capacity was 20 TBs. The internode crossbar network had 10 TB/s bandwidth. The building in which the ES was housed was purpose built, and included lightening suppression and protection against earthquakes.

Pipelining is among the most widely employed and enduring forms of parallelism. It is so broad in applicability that it goes far beyond its use in computing and impacts many aspects of our daily lives. Mass production of automobiles is an example. Each frame moves from station to station to have one small assembly action (e.g., attaching a side mirror) performed on it. The vehicle then moves to the next station to have yet another assembly step performed, while at the previous station a new car frame rolls into place to get its mirror. Many different cars are being built at the same time on the assembly line, and it can take a long time for a car to be built—from hours to days. But the miracle is that a new car is finished every few minutes and driven out of the manufacturing plant.

Pipelining is used in many places within supercomputing system architecture, even today. At one time it was the principal form of parallelism exploited in a class of supercomputers referred to as vector computers, of which the Cray-1 [7] launched in 1975 is perhaps the archetype and recognized as the iconic supercomputer.

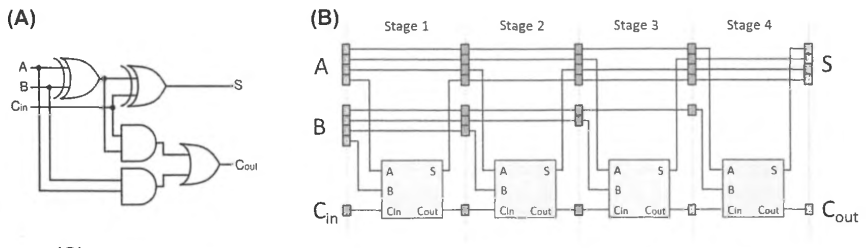
2.6.1 PIPELINE PARALLELISM

Pipeline parallelism is derived by dividing a complicated action into a sequence of simpler actions, each of which may be performed independently. For any one instance of the complex action, only one stage is performed at any one time and there is no concurrency. But when many instances of the same action have to be performed, they can be issued to the pipeline one at a time so that in each pipeline stage one of the concurrent operations is being performed. Thus a complex action is divided into a sequence of simpler steps and different parts of multiple operations are performed simultaneously. It is the combination of the number of separate instances of a given operation that need to be performed and the number of pipeline stages in which the operation may be divided that yields the parallelism. Further, exploitation of this kind of parallelism, as is the case for many other forms of parallelism, requires a combination of hardware architecture that can do many different things at the same time and software that exposes and controls the application parallel work to be performed. Thus except in special cases, both hardware and software have to work together to exploit this kind of parallelism.

A simple example will illustrate these basic ideas. We propose to increment (add one) to every integer of a block of numbers. To do this very fast we want to overlap the carries between successive bits (here four bits for ease of exposition). Thus the time to perform each stage is only that of a single bit full add rather than having to wait for the propagation of carries through the entire sequence of bits. This is illustrated in Fig. 2.10. Every stage of the addition pipeline consumes one bit of each input operand and generates a corresponding bit of output sum, starting with the least significant bit (bit 0). A dedicated chain of registers implements the intermediate storage for the carry bit. As the operation progresses, the input bits of operand A are successively replaced with the computed sum bits, finally resulting in the full sum and fours-bit addition carry bit produced at the end of stage 4.

The effective operation of a pipeline is a function of five parameters:

- t_s: operation time of each pipeline stage
- t_v: overhead to switch between successive stages
- p_s: number of stages
- n_d: number of input datasets
- t_m: execution time of a monolithic version of the same function.



(C)

Register	Input	Stage 1 output	Stage 2 output	Stage 3 output	Stage 4 output
Operand A / Sum	$A_3A_2A_1A_0$	$A_3A_2A_1S_0$	$A_3A_2S_1S_0$	$A_3S_2S_1S_0$	$S_3S_2S_1S_0$
Operand B	$B_3B_2B_1B_0$	$B_3B_2B_1$	B_3B_2	B_3	
Carry	C_{in}	C_{0out}	C_{1out}	C_{2out}	C_{3out}

FIGURE 2.10

(A) Single-bit full adder. (B) Complete pipeline with input data, operation, and output. (C) Propagation of computations through individual pipeline stages.

The operation time of a pipeline stage is the propagation delay through the digital logic of the stage from when the input data is presented to it until it delivers the resulting data of that stage. The overhead is the small amount of time needed to control the movement of data from the output of a previous stage to the input of the next stage. Different kinds of functions can be divided into different number of parallel pipeline stages. Even for a given function there may be many alternative ways to divide the required workload into multiple successive stages. To determine the speedup of a pipeline structure, G, the total throughput rate is calculated and compared with the alternative monolithic structure.

$$G = \frac{T_m}{T_p} \tag{2.3}$$

$$T_m = t_m \times n_d \tag{2.4}$$

$$T_p = (t_v + t_s) \times (p_s + n_d),\ p_s > 1 \tag{2.5}$$

$$G = \frac{t_m \times n_d}{(t_v + t_s) \times (p_s + n_d)} \tag{2.6}$$

The total time to execute a set of data operands, T_m, employing a monolithic logical functional unit is simply the product of the time to perform the calculation of a single dataset and the number of such datasets to be processed. No parallelism is being used or exploited, except perhaps at the bit level within the logic design. The total time to execute all the sets of data operands, T_p, by a pipelined structure of logical stages is more complex. The finest-grain time is that of the propagation delay through a given stage, t_p, to which must be added a small amount of time to coordinate between successive stages and transfer partial results and operand values from each stage to its succeeding one, t_v, which is referred to as overhead time. The total number of steps to perform the complete computation for all the datasets is a function of both the number of such datasets, n_d, and the number of stages in the pipeline, p. It can be viewed as the number of steps to fill the pipeline with data plus the number of steps to empty the pipeline, which is the number of data elements.

The advantage achieved through pipelining or its performance gain, G, is given as the ratio of the monolithic logic execution time of all the data and the time required for the pipelined structured to complete the operation on the same argument dataset as presented in the equation above. A successful pipeline structure is one for which the following conditions hold:

$$\begin{gathered} t_p \ll t_m \\ (p \times t_p) > t_m \\ n_d \gg p \\ t_p \gg t_v \end{gathered} \tag{2.7}$$

Under these favorable conditions, the limit is:

$$\lim_{n_d \to \infty} G \cong \frac{t_m}{t_p} \tag{2.8}$$

This returns the optimal performance gain for a structure exploiting pipelined parallelism (Fig. 2.11). However, there are boundary conditions that limit the degree to which the performance gain can be increased through reduction of size to t_p. Major constraints are as follows.

1. The number of logic layers within a pipeline stage cannot be practically reduced below four- to six-gate depth, thus imposing a lower bound of a few gate delays.
2. The overhead, t_v, imposes a second bound on G when $t_v \geq t_p$.

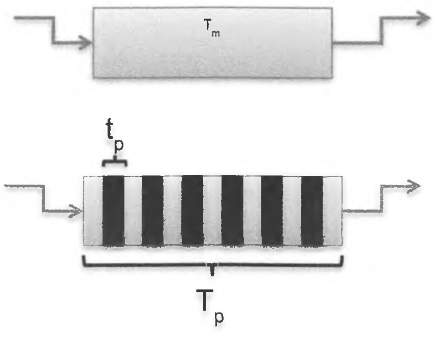


FIGURE 2.11

Pipeline parallelism.

3. The slowest pipeline stage (longest propagation delay) determines the clock rate of the entire pipeline.
4. When n_d becomes small with respect to p.

Performance gain is bounded by the overhead, t_v, limiting the degree of useful pipelining that can be achieved.

$$\lim_{p \to \infty} G = \frac{t_m}{t_v}, \quad \text{for } n_d \to \infty \tag{2.9}$$

While the propagation delay of a pipeline stage is much less than that of the monolithic logic version of the same function, the advantage of pipelining the logic is lost when n_d is very small or even length 1 (scalar). In this case, performance gain could be less than 1. The breakeven point for the special case of no overhead ($t_v = 0$) is:

$$n_d \geq \frac{p \times t_p}{t_m - t_p} \tag{2.10}$$

With $t_p \ll t_m$, this converges to: $n_d \geq p \times t_p/t_m$.

Pipeline structures are employed in many aspects of computer architecture. As is described later, a common application is in the execution pipeline, where many instructions can be executing at the same time, significantly increasing the rate of instruction throughput. However, the iconic use of pipelining for supercomputing was in the vector-processing architecture of the mid-1970s, described in the next subsection.

2.6.2 VECTOR PROCESSING

Vector-processing architecture exploits pipelining to achieve the advantages of fine-grain parallelism, latency hiding, and amortized control overheads (Fig. 2.12). Pipelining also permits a high clock rate for vector-based computer architecture by keeping the pipeline stages small in terms of the logic depth.

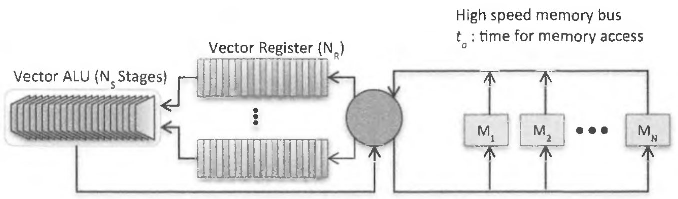


FIGURE 2.12

The vector-processing architecture.

2.7 SINGLE-INSTRUCTION, MULTIPLE DATA ARRAY

The SIMD array was a major class of parallel computer architecture in the 1980s and 1990s. It was particularly well suited to LSI technology, although such systems were implemented both before and after this era. This strategy for parallel processing is still found in subsystems of a wide range of computers today for specialized tasks and accelerators.

2.7.1 SINGLE-INSTRUCTION, MULTIPLE DATA ARCHITECTURE

The SIMD array class of parallel computer architecture consists of a very large number of relatively simple PEs, each operating on its own data memory (Fig. 2.13). The PEs are all controlled by a shared sequencer or sequence controller that broadcasts instructions in order to all the PEs. At any point in time all the PEs are doing the same operation but on their respective dedicated memory blocks. An interconnection network provides data paths for concurrent transfers of information between PEs, also managed by the sequence controller. I/O channels provide high bandwidth (in many cases) to the system as a whole or directly to the PEs for rapid postsensor processing. SIMD array architectures have been employed as standalone systems or integrated with other computer systems as accelerators.

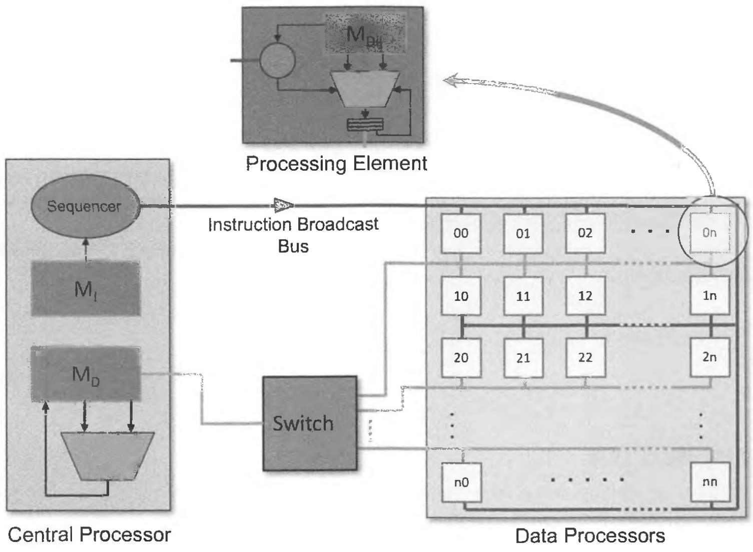


FIGURE 2.13

The SIMD array class of parallel computer architecture.

The PE of the SIMD array is highly replicated to deliver potentially dramatic performance gain through this level of parallelism. The canonical PE consists of key internal functional components, including the following.

- Memory block—provides part of the system total memory which is directly accessible to the individual PE. The resulting system-wide memory bandwidth is very high, with each memory read from and written to its own PE.
- ALU—performs operations on contents of data in local memory, possibly via local registers with additional immediate operand values within broadcast instructions from the sequence controller.
- Local registers—hold current working data values for operations performed by the PE. For load/ store architectures, registers are direct interfaces to the local memory block. Local registers may serve as intermediate buffers for nonlocal data transfers from system-wide network and remote PEs as well as external I/O channels.
- Sequencer controller—accepts the stream of instructions from the system instruction sequencer, decodes each instruction, and generates the necessary local PE control signals, possibly as a sequence of microoperations.
- Instruction interface—a port to the broadcast network that distributes the instruction stream from the sequence controller.
- Data interface—a port to the system data network for exchanging data among PE memory blocks.
- External I/O interface—for those systems that associate individual PEs with system external I/O channels, the PE includes a direct interface to the dedicated port.

The SIMD array sequence controller determines the operations performed by the set of PEs. It also is responsible for some of the computational work itself. The sequence controller may take diverse forms and is itself a target for new designs even today. But in the most general sense, a set of features and subcomponents unify most variations.

As a first approximation, Amdahl's law may be used to estimate the performance gain of a classical SIMD array computer. Assume that in a given instruction cycle either all the array processor cores, p_n, perform their respective operations simultaneously or only the control sequencer performs a serial operation with the array processor cores idle; also assume that the fraction of cycles, f, can take advantage of the array processor cores. Then using Amdahl's law (see Section 2.7.2) the speedup, S, can be determined as:

$$S = \frac{1}{1 - f + \left(\frac{f}{p_n}\right)} \tag{2.11}$$

2.7.2 AMDAHL'S LAW

Ideally, all parts of a computation from beginning to end could be further partitioned into parallel pieces executing concurrently, such that many computing resources could be applied to the computation simultaneously to reduce time to solution uniformly (across the computation) and accelerate the rate of processing. While there are some extreme examples of this ideal case, more frequently application programs exhibit operational behavior such that some parts of the computation are indeed parallel, supporting acceleration through concurrent operation, while other parts show less or even no parallelism in the limit, operating purely sequentially and issuing only one instruction at a time. This computing profile combining both parallel and sequential fractions of the complete execution imposes an important bound on the maximum acceleration that can be achieved through parallelism. The most widely recognized formulation of this boundary condition is referred to as Amdahl's law after the famous computer architect who first codified it. While more broadly applicable to parallel computing in general, Amdahl's law is particularly well suited to modeling the performance of SIMD array computing (with some slight simplifying assumptions), and this motivates its introduction here. Later it is employed to understand other forms of parallel architectures.

Assume that a SIMD array has two modalities of execution, either sequential, where its central processor performs one instruction at a time, or parallel, where all the array processor cores perform their respective operations at the same time. For simplicity it is assumed that the clock rates of both the central processor and the array processors are the same, but this is of little importance to the implications of this performance model.

Figs. 2.14 and 2.15 show two timelines: the first a sequential execution of all the operations of a computation T_0, and the second with the fraction f of the operations, T_f, done in parallel, with a level of parallelism g. Ideally, the gain in performance would be g. But only T_F of the total T_0 operations can be performed in parallel to at least some degree, with the remaining $T_0 - T_F$ operations still done sequentially, where $f = T_F/T_0$. As shown below, the actual speedup, S, can be determined by the ratio of the times of the two solution times with and without g parallelism, such that $S = T_0/T_A$ where $T_A = T_F/g$. The resulting formulation for S is derived below as a function of g and f, independent of the exact times involved.

$$S = T_0/T_A \tag{2.12}$$

$$f = T_F/T_0 \tag{2.13}$$

$$T_A = (1-f) \times T_0 + \left(\frac{f}{g}\right) \times T_0 \tag{2.14}$$

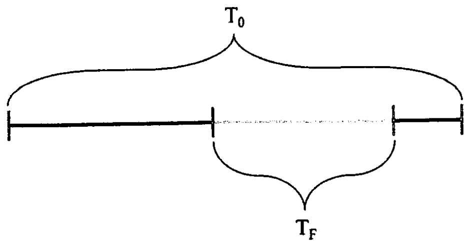


FIGURE 2.14

The timeline of a computation performed sequentially taking time T_0. The black segments of the line of execution represent the set of operations that have to be done in order. The green (light gray in print versions) segment of the line of execution represents the set of T_F operations that can be performed concurrently.

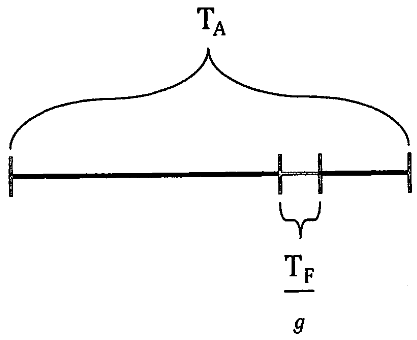


FIGURE 2.15

The timeline of a computation where those operations that can be done in parallel are performed concurrently with a parallelism of g, resulting in a shorter time to solution, T_A.

$$S = \frac{T_0}{(1-f) \times T_0 + \left(\frac{f}{g}\right) \times T_0} \tag{2.15}$$

$$S = \frac{1}{1-f+\left(\frac{f}{g}\right)}, \tag{2.16}$$

where $T_0 \equiv$ time for nonaccelerated computation; $T_A \equiv$ time for accelerated computation; $T_F \equiv$ time of portion of computation that can be accelerated; $g \equiv$ peak performance gain for accelerated portion of computation; $f \equiv$ fraction of nonaccelerated computation to be accelerated; $S \equiv$ speedup of computation with acceleration applied.

This formulation of Amdahl's law can be understood by considering various possible operating points. In the limits, if the entire code can be executed (equally) in parallel by a concurrency of g, then the fraction $f = 1$ and the total speedup is the ideal case of g. But if none of the code can be performed in parallel despite having g-level hardware parallelism, $f = 0$ and $S = 1$, so there is no gain as one would expect. What is more sobering, and why Amdahl's law is so important, is seen with yet another operating point. Suppose the parallel hardware exhibits an ideal gain of 1 million, that is $g = 1{,}000{,}000$, and half the code can be done concurrently, $f = 0.5$. Simple substitution shows that in spite of this enormous potential gain, the actual delivered speedup is less than 2; $S < 2$. In fact, even if g were infinity, with a fraction of 0.5, you would still not get a speedup greater than 2. A range of speedups is shown in Fig. 2.16 with respect to the fraction of total time to be accelerated with different ideal accelerator gains.

Example. Consider a SIMD array computer with an 8×8 array of core processors and one sequential control processor. In a given computing cycle, either the control processor performs an operation or all the array cores perform the same operation on their respective data. What fraction of the total workload needs to be performed by the core array to deliver an overall speedup of eight times?

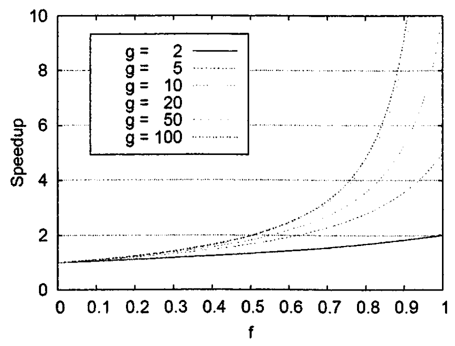


FIGURE 2.16

Examples of delivered speedup with respect to the fraction of code that can be parallelized and the gain of the accelerator hardware.

The total speedup, $S = 8$, is required with an acceleration gain, $g = 64$. Substituting appropriately:

$$S = \frac{1}{1 - f + \frac{f}{g}}$$

$$8 = \frac{1}{1 - f + \frac{f}{64}}$$

$$8 - 8f + \frac{f}{8} = 1$$

$$64 - 63f = 8$$

$$56 = 63f$$

$$f = \frac{56}{63} = 0.889$$

here it is seen that to achieve only 12.5% of the peak gain of 64 requires almost 90% of the workload to be able to be parallelized. Experience throughout this course will demonstrate that this is a tall order and difficult to achieve.

2.8 MULTIPROCESSORS

The multiprocessor class of parallel computer is the dominant form of supercomputer today. Most broadly, it is any system comprising a set of individual self-controlled computers integrated by a communications network and coordinated to perform a single workload. By the Flynn taxonomy the multiprocessor is a MIMD-class machine. Each processor making up the system has its own data-processing units controlled by its own local instruction stream controller. Multiprocessors have a long history reaching back to the 1950s and SAGE, consisting of MIT Whirlwind-class computers deployed by IBM for the US Air Force at North American Aerospace Defense Command. A number of commercial multiprocessors consisting of two processors were deployed, with one processor dedicated to the computing (heavy lifting) while the other managed I/O tasks.

Multiprocessors grew in importance for supercomputing with the advent of VLSI technology and the development of microprocessor architecture. This represented an important change in the trend and direction of supercomputer architecture. Cost benefits through the exploitation of economy of scale derived from the mass market of general-purpose microprocessors defined the next generation of high performance computers. The integration of microprocessors derived for the broader markets of workstations, personal computers, and enterprise servers as the principal compute engines of supercomputers had a dramatic impact on large-scale system architecture, largely displacing the previous specialized designs. One visible impact was the sheer physical size of supercomputers, which greatly escalated to multiple rows of racks, each incorporating many VLSI microprocessors. Today the multiprocessor has made yet another leap in technology, with a decade of multicore technology where each socket now incorporates multiple processors, referred to as "cores".

There are three mainstream configurations in use: SMPs, MPPs, and commodity clusters. A single processor system reflects a unified memory in which all of your data sits in the same memory subsystem. When multiple processors are used, a choice has to be made about how the processors and memory are interrelated. Do all the processors within the system share the same memory subsystem, or does each processor have its own separate memory? A third choice is somewhere in between: groups of processors share a memory block while the different groups, often referred to as "nodes", have distinct memory blocks. With multicore sockets this last is often the structure employed. These different classes of multiprocessor system architectures are described in detail in the following subsections.

2.8.1 SHARED-MEMORY MULTIPROCESSORS

A shared-memory multiprocessor is an architecture consisting of a modest number of processors, all of which have direct (hardware) access to all the main memory in the system (Fig. 2.17). This permits

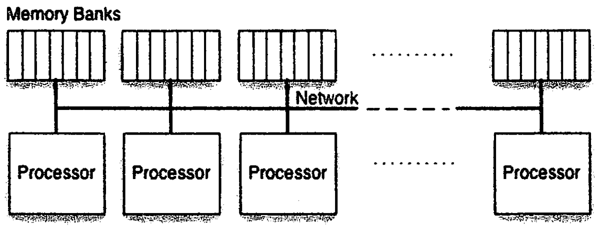


FIGURE 2.17

The shared-memory multiprocessor architecture.

any of the system processors to access data that any of the other processors has created or will use. The key to this form of multiprocessor architecture is the interconnection network that directly connects all the processors to the memories. This is complicated by the need to retain cache coherence across all caches of all processors in the system.

Cache coherence ensures that any change in the data of one cache is reflected by some change to all other caches that may have a copy of the same global data location. It guarantees that any data load or store to a processor register, if acquired from the local cache, will be correct, even if another processor is using the same data. The interconnection network that provides cache coherence may employ any one of several techniques. One of the earliest is the modified exclusive shared invalid (MESI) protocol, sometimes referred to as a "snooping cache", in which a shared bus is used to connect all processors and memories together. This method permits any write of one processor to memory to be detected by all other processors and checked to see if the same memory location is cached locally. If so, some indication is recorded and the cache is either updated or at least invalidated, such that no error occurs.

Shared-memory multiprocessors are differentiated by the relative time to access the common memory blocks by their processors. A SMP is a system architecture in which all the processors can access each memory block in the same amount of time. This capability is often referred to as "UMA" or uniform memory access. SMPs are controlled by a single operating system across all the processor cores and a network such as a bus or cross-bar that gives direct access to the multiple memory banks. Access times can still vary, as contention between two or more processors for any single memory bank will delay access times of one or more processors. But all processors still have the same chance and equal access. Early SMPs emerged in the 1980s with such systems as the Sequent Balance 8000. Today SMPs serve as enterprise servers, deskside machines, and even laptops using multicore chips, and thus play a major role in the medium-scale computing which is a major part of the commercial market. SMPs also serve as nodes within much larger MPPs.

Nonuniform memory access (NUMA) architectures retain access by all processors to all the main memory blocks within the system (Fig. 2.18). But this does not ensure equal access times to all

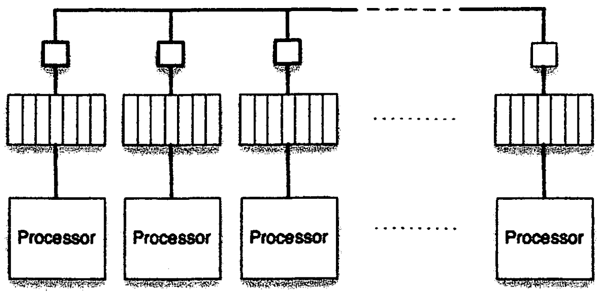


FIGURE 2.18

Nonuniform memory access architectures retain access by all processors to all the main memory blocks within a system, but does not ensure equal access times to all memory blocks by all the processors.

memory blocks by all processors. This is motivated by the architecture opportunity provided by modern microprocessor designs to exploit high-speed local memory communication channels while providing access to all the memory through external, albeit slower, global interconnection networks. NUMA architectures benefit from scaling, permitting more processor cores to be incorporated into a single shared-memory system than SMPs. However, because of the difference in memory access times, the programmer has to be conscious of the locality of data placement and use it to take best advantage of computing resources. NUMA multiprocessor architectures first emerged with such systems as the BBN Butterfly multiprocessors, including the GP-1000 and the TC-2000.

2.8.2 MASSIVELY PARALLEL PROCESSORS

MPP architecture is the structure that most easily scales to the extremes of computing system size and performance (Fig. 2.19). The largest supercomputers today, comprising millions of processor cores, are of this class of multiprocessor. MPPs are (in most cases) not shared-memory architectures, but are distributed memory. In an MPP separate groups of processor cores are directly connected to their own local memory. Such groups are colloquially referred to as "nodes", and there is no sharing of memory between them; this simplifies design and eliminates inefficiencies that impede scalability. But in the absence of shared memories, a processor core in one group must employ a different method to exchange data and coordinate with cores of other processor groups. The logical capability for message passing is enabled by the physical system area network (SAN) that integrates all the nodes to form a single system. As discussed in greater detail in Chapter 8, a message is transferred between two processor cores of the system, with each core running a separate process. By this means a receiving process and its host processor can acquire data from a sending processor's process. The same network can be used to synchronize processes running on separate processors. By 1997 the first system capable of teraflops (HPL benchmark) was the Intel ASCI Red MPP deployed at Sandia National Laboratories.

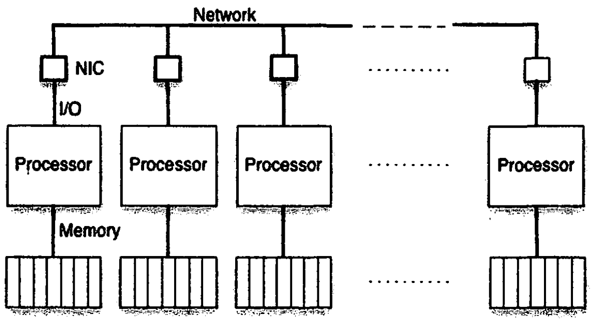


FIGURE 2.19

The massively parallel processor class of parallel computer architecture.

2.8.3 COMMODITY CLUSTERS

While all current generations of supercomputers exploit the economic advantages of incorporating VLSI microprocessors and DRAM main memory that are mass produced for commercial and consumer markets, the systems discussed thus far are still based on special-purpose designs to provide tight coupling among processor cores for superior performance. However, the dominant class of deployed supercomputers, the commodity clusters, take exploitation of mass-market economics one step further to introduce even an greater cost advantage. As the name suggests, such systems consist exclusively of commodity subsystems, sometimes referred to as COTS (commodity off-the-shelf) components. Dongarra et al. [8] defines a "commodity cluster" as "a cluster in which both the network and the compute nodes are commercial products available for procurement and independent application by organizations (end users or separate vendors) other than the original equipment manufacturer". The key idea is that a supercomputer can be made up of component subsystems, all of which can be procured by and are produced for a much larger user market than the deployed base of supercomputers, thus leveraging economy of scale for dramatic improvements of performance to cost.

Emerging in the mid-1990s, such improvements often exceeded an order of magnitude when using consumer-grade system components. In 1997 the network of workstations (NOW) cluster [9] of commercial-grade workstations from the University of California at Berkeley was the first commodity cluster to be placed in the Top 500 list, and the Beowulf cluster [10] (of consumer-grade personal computers) from NASA Jet Propulsion Laboratory and the California Institute of Technology was the first to be awarded the Gordon Bell Prize (Fig. 2.20). As this early history suggests, there were two levels of commodity clusters due to two corresponding levels of microprocessors. Commercial-grade microprocessors were used for industrial-grade workstations where performance mattered, while consumer-grade microprocessors were used for personal computers where cost was of the greatest importance. Eventually this differentiated market merged, as 32-bit and eventually 64-bit architectures became common.

FIGURE 2.20

An example of the Beowulf class of parallel computer architecture.

Typically commodity clusters exhibit lower efficiencies with respect to number of cores than MPPs. While contemporary MPPs may demonstrate efficiency approaching 90% on some workloads, commodity clusters are more likely to yield efficiency between 60% and 70% (HPL benchmark). However, workloads vary significantly in terms of degree of coupling, and clusters are excellent for throughput computing such as parameter sweeps that rely less on intercommunication and more on local processing capabilities.

Today the fastest supercomputers are a mix of MPPs and commodity clusters. Clusters currently exceed 80% (four out of five) of all the systems rated by the Top 500 list. Not surprisingly, due to the superior properties of the MPPs purpose built for supercomputing with their optimized SANs, the majority of the fastest machines are of this class. Historically commodity clusters have employed a number of COTS networks, including such early offerings as asynchronous transfer mode (ATM), Myrinet, and Ethernet 100BaseT. Current-generation clusters principally employ Gigabit Ethernet or Infiniband Networks.

2.9 HETEROGENEOUS COMPUTER STRUCTURES

The homogeneous computing systems described so far employ a single type of processing component to perform all computation, such as a typical multicore processor socket component. The majority of supercomputers are of this type. However, for certain patterns of computing, other core designs and structures made of them can deliver sometimes dramatic performance improvements, at least for some kinds of computing algorithms. Systems comprising two or more types of computer cores, sockets, and nodes are distinguished from homogeneous computing systems that have only one type, and are designated as heterogeneous systems. Accelerators, sometimes known as GPUs, are attached to a system node via the I/O bus, principally the peripheral component interconnect bus, and can be accessed by any of the conventional processor cores of the system within the same node. Accelerators are designed to perform certain classes of computation extremely well, like linear algebra and signal processing problems. Heterogeneity is also finding its way directly into chip design, thus circumventing the intermediary I/O bus.

While each of these classes of architecture, both sequential and multiple forms of parallel, has been presented as separate and distinct, modern computer architecture such as MPPs in the broadest sense often incorporate the best aspects of all of them. A modern microprocessor socket incorporates structures derived from each of these main types, including sequential, pipelining, SIMD, and multiprocessor organizations.

2.10 SUMMARY AND OUTCOMES OF CHAPTER 2

- Computer architecture is the structure and semantics of a computer. HPC architecture is optimized to achieve high speed through aggressive exploitation of fast technologies and parallel organization of its component modules.

- Key properties of HPC architecture include speed of operation, parallelism for doing multiple operations at the same time, efficient use of critical components, the electrical power that it consumes, reliability, and how easy it is to program.
- Flynn's taxonomy of parallel architectures, while a bit stale, is still widely cited. It includes SISD, SIMD, and MIMD, which can be applied to system types today.
- HPC has advanced over many generations, with progress determined in part by evolving device technologies, including simple devices (e.g., abacus), mechanical gears, electromechanical, such as the Hollerith, and electronics, including vacuum tubes, transistors, and integrated circuits.
- Technologies serve multiple purposes. These include storing information in binary (base 2) form in memory (e.g., magnetic cores and DRAM), performing operations using Boolean logic, and moving data on buses and network interconnect channels.
- The von Neumann architecture is the foundational concept for sequential stored-program computers, which are the basis for essentially all modern supercomputers today.
- HPC systems are derivatives of the von Neumann architecture through many innovations, which include diverse forms of parallelism, memory hierarchies, and advanced networks to integrate the many subsystems together.
- Pipelining connects successive component stages together so data can pass from one stage to the next for rapid throughput.
- Vector architectures (e.g., the Cray-1) exploit pipelining for high-speed arithmetic units, register loads and stores, and overlapping memory accesses.
- SIMD array processing uses many lightweight cores dedicated to separate partitioned memory banks. All cores perform the same kind of operation at the same time but on their local data, to achieve a high degree of parallelism managed by a control processor.
- MPPs are single systems comprising many integrated computer processors. There are diverse forms of multiprocessor, differentiated by the way they are interconnected and the relationship between processors and memory banks.
- Shared-memory multiprocessors combine individual processors with multiple memory banks, such that all processors are able to access all the shared memory banks. SMPs have equal access to all memory in terms of time and bandwidth. Distributed shared-memory (DSM) architectures also share the same memories but have preferential access to some memory banks in lieu of others. SMPs are referred to as UMA, while DSMs exhibit NUMA behavior.
- Commodity clusters are another form of multiprocessor, made entirely from subsystems that are COTS to exploit economy of scale for superior performance to cost.
- Amdahl's law relates achievable delivered speedup to the gain of a parallel accelerator and the fraction of the total workload that can be performed in parallel.

2.11 QUESTIONS AND PROBLEMS

1. Define or expand each of the following terms or acronyms.

• Computer architecture	• SISD	• Punched cards
• ISA	• SIMD	• Paper tape
• Parallelism	• MIMD	• Cycle time
• GHz	• MISD	• Volatile
• DRAM	• SPMD	• Destructive read
• SRAM	• GPU accelerator	• Register
• NVRAM	• Abacus	• Accumulator
• Cache	• Harvard architecture	• PE
• Bandwidth	• Chip	• Interconnection network
• Latency	• Mainframe	• Data path
• NRE	• Minicomputer	• I/O channel
• Botzmann constant	• Vector processor	• Amdahl's law
• Data path	• SIMD array	• Shared-memory multiprocessor
• Control path	• CMOS	• Cache coherence
• Efficiency, floating-point efficiency	• SMP	• MESI protocol
• Hardware fault	• MPP	• Snooping cache
• Software fault	• Vacuum tube	• SMP
• Hard fault, soft fault	• Discrete transistor	• UMA
• Cosmic ray	• Integrated transistor	• NUMA
• Checkpoint/restart	• Logic gate	• SAN
• Programmability	• Circuit	• COTS
• SLOC	• Switching rate	• ATM
• Programming model	• Propagation delay	• Myrinet
• Productivity	• Mercury tank	• Gigabit Ethernet
• Software engineering	• One-dimensional memory	• Infiniband
• Workflow management	• Two-dimensional memory	• Heterogeneous system architecture
• Flynn's taxonomy, Michael Flynn	• Core memory	• Microprocessor socket

2. State whether each of the following statements is true or false.

- HPC architecture is concerned with only the lowest-level technologies and circuit design.
- An HPC system will never be fast enough to deliver the necessary performance required by major application problems.

- Design an effective automotive assembly strategy that includes a pipeline. Analyze your design, including a calculation of the throughput of your system and the performance gain of your system over a simple monolithic assembly system. State your assumptions explicitly in your analysis.

17. Name and describe the key internal functional components of an SIMD array.

18. Suppose that 5% of my program is sequential and cannot be parallelized. If I can execute my program in 10 min on one processor, how fast can I expect it to run on 10 processors according to Amdahl's law? What is the maximum speedup that I can obtain, no matter how many processors I use, according to Amdahl's law.

19. Discuss the efficiency that is likely obtainable with commodity clusters versus MPPs. How can workload affect efficiency?

REFERENCES

[1] M.W. Wilkes, W. Renwick, The EDSAC (electronic delay storage automatic calculator), Mathematics of Computation 4 (1950) 61–65.

[2] IBM Corp, 704 Data Processing System, August 30, 2013 [Online]. Available: http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP704.html.

[3] J. von Neumann, First Draft of a Report on the EDVAC, University of Pennsylvania, 1945.

[4] Institute for Advanced Study, IAS Electronic Computer Project, 2017 [Online]. Available: https://www.ias.edu/electronic-computer-project.

[5] Massachussets Institute of Technology, Project Whirlwind, MIT Institute Archives & Special Collections, 2008 [Online]. Available: https://libraries.mit.edu/archives/exhibits/project-whirlwind/.

[6] I.B.M. Corp., 1620 Data Processing System, August 30, 2013 [Online]. Available: https://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP1620.html.

[7] Cray Research, Inc., Cray-1 Computer System Hardware Reference Manual, 1977 [Online]. Available: http://history-computer.com/Library/Cray-1_Reference%20Manual.pdf.

[8] J. Dongarra, T. Sterling, H. Simon, E. Strohmaier, High-performance computing: clusters, constellations, MPPs, and future directions, Computing in Science & Engineering 7 (2) (2005) 51–59.

[9] T.E. Anderson, D.E. Culler, D.A. Patterson, A case for NOW (networks of workstations), IEEE Micro 15 (1) (1995) 54–64.

[10] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, C.V. Packer, BEOWULF: a parallel workstation for scientific computation, in: In Proceedings of the 24th International Conference on Parallel Processing, 1995.

- The time to do a floating-point operation is the most important aspect of the efficiency of an HPC system.
- The cost of the software on an HPC system is much less than the cost of the hardware platform.
- The greater the performance that is required, the harder it is to optimize the user program.
- Code reuse is critical to managing application development complexity and difficulty.
- Magnetism is the longest-lasting technology in computing, having been used from the 1940s to the present (2010s).

3. HPC architecture exploits enabling technologies to minimize ____, maximize ____, and serve ______. In recent years HPC has been applied to ______.
4. Name and describe the four principal sources of performance degradation. In addition to performance, name and describe four factors that are of concern for HPC.
5. Explain what distinguishes a supercomputer from a commercial or consumer-grade server.
6. Name and describe three key properties that determine the performance of an HPC architecture. Give a formula that shows the relationship of these properties. Name and describe two additional factors that influence HPC systems operation.
7. Which has better performance, a supercomputer with 10,000 CPUs that can operate in parallel at a clock rate of 2.9 GHz and is available 80% of the time, or a supercomputer with 10,000 CPUs that can operate in parallel at a clock rate of 2.7 GHz and is available 95% of the time?
8. Name and describe three aspects of speed of an HPC system. What are the two measures for communication speed?
9. Describe how air cooling works. Explain why liquid cooling is sometimes used.
10. Describe three types of faults in an HPC system. Describe how recovery from a fault can work.
11. Describe two modern architecture strategies that address the two trends of stagnant processor speed and multicore chips.
12. Name and briefly describe the three dominant classes of technologies that define the design space for HPC architecture.
13. Describe the purpose of cache memory. Describe the purpose of L1, L2, and L3 cache.
14. Discuss at least three different features of SRAM, DRAM, and NVRAM, including their relative access speeds.
15. Draw a figure that illustrates the principal elements of the von Neumann architecture.
16. Suppose that you want to optimize an automotive assembly plant. In your plant the automotive frame and whole components arrive at the plant and the task is to assemble the complete automobile. For simplicity, let us assume there are five steps to assembling an automobile: add and configure engine in frame; add seats; assemble wheel subsystem; assemble steering; and assemble and configure braking system. Also for simplicity, assume that these can be done in any order and can be done at the same time, except that the assembly of the seats cannot happen at the same time as the assembly of the steering system, and assembly of the brakes has to come after the assembly of the wheels. Also assume that the time for all steps is the same, except that the time to assemble the braking system is two times the time required for each of the other steps.
    - Discuss at least three alternatives that are possible for an automotive assembly strategy. Include SISD, SIMD, MISD, and MIMD alternatives in your discussion.

CHAPTER

COMMODITY CLUSTERS

3

CHAPTER OUTLINE

3.1 Introduction **84**
3.1.1 Definition of "Commodity Cluster" 84
3.1.2 Motivation and Justification for Clusters 84
3.1.3 Cluster Elements 85
3.1.4 Impact on Top 500 List 86
3.1.5 Brief History 88
3.1.6 Chapter Guide 90
3.2 Beowulf Cluster Project **91**
3.3 Hardware Architecture **93**
3.3.1 The Node 93
3.3.2 System Area Networks 94
3.3.3 Secondary Storage 95
3.3.4 Commercial Systems Summary 95
3.4 Programming Interfaces **97**
3.4.1 High Performance Computing Programming Languages 97
3.4.2 Parallel Programming Modalities 97
3.5 Software Environment **98**
3.5.1 Operating Systems 98
3.5.2 Resource Management 99
3.5.3 Debugger 101
3.5.4 Performance Profiling 101
3.5.5 Visualization 101
3.6 Basic Methods of Use **104**
3.6.1 Logging On 104
3.6.2 User Space and Directory System 105
3.6.3 Package Configuration and Building 110
3.6.4 Compilers and Compiling 112
3.6.5 Running Applications 113
3.7 Summary and Outcomes of Chapter 3 ...... **113**
3.8 Questions and Exercises **114**
References **114**

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00003-4

3.1 INTRODUCTION

The commodity cluster [1] represents possibly the single most successful form of supercomputing in the history of high performance systems. It exploits technology advancements in the areas of very large-scale integration microprocessors, dynamic random access memory (DRAM), and networking, as well as improving performance relative to cost through the economy of scale of mass production. The secret of the success of commodity clusters is that they comprise major components that are standalone computers in their own right, marketed to a much larger market segment than the narrower high performance computing (HPC) community, and delivering an economy of scale benefit from the larger consumer base. But the commodity cluster is also successful because it provides great flexibility of system configuration and tremendous accessibility for a broad range of users, from the most arcane national laboratory scientists to high-school students. Indeed, commodity clusters may constitute the third revolution in supercomputing since its inception in the late 1940s. For our purposes, the commodity cluster will serve as the archetype of the conventional scalable HPC system, and its description will delineate the many system component layers that comprise a full supercomputer in terms of both hardware and software.

3.1.1 DEFINITION OF "COMMODITY CLUSTER"

The commodity cluster is a group of integrated computer systems. The component computers are standalone, capable of independent operation, and marketed to a much broader consumer base than the scaled clusters which they comprise. The integration network employed is separately developed and marketed for use by a systems integrator. Mass storage devices are off the shelf and either physically installed within the system nodes or connected externally. All interfaces adhere to industry standards for both attached devices (e.g., USB [2], peripheral component interconnect express [PCIe] [3]) and system networking. Although not required, system software is usually open-source, nonproprietary, and Linux based. Programming interface libraries are bound to C, C++, or Fortran and employ a message-passing interface (MPI), OpenMP, or both.

3.1.2 MOTIVATION AND JUSTIFICATION FOR CLUSTERS

While the motivation for adopting commodity clusters may differ among individuals and institutions, there are clear attributes that have justified their adoption for the last 2 decades. Some of the most prevalent among these are briefly discussed below.

Accessibility—more likely than not, access to a medium-scaled supercomputer will link to a commodity cluster of moderate scale. This is in part because they are by far the most prevalent form of supercomputer available. But it is also true that due to their cost, commodity clusters make up a disproportionate number of systems deployed at institutions where entry-level experience is likely to be acquired.

Performance relative to cost—the relatively low cost of acquisition and within some environments the low cost of ownership are often the dominant reason for the procurement choice of commodity clusters. Simply put, one gets more peak performance for a given price than using more tightly coupled, admittedly more efficient alternatives.

memory access and processor to I/O controllers, and the node "chip-set" that manages the node transparently to the user while providing basic primitives for the OS and bootstrapping the node from the powered-down state. The node may include a diversity of I/O controllers for various purposes, including but not limited to the system area network(s) (SANs).

- The system area network, or SAN—the off-the-shelf communication channel that interconnects all the nodes together into a single distributed computing system. The network supports data message passing between nodes, interprocess synchronization such as global barriers, and other collective actions such as reduction operations. It also may support communication with external system I/O devices and the internet. The network consists of the physical data paths of either copper wires or fiber optics, network interface controllers (NICs) to move node data to the data paths, and routers for switching data between data paths to arrive at the destination node.
- Host—a special node to support user services, including login accounts, administration, resource allocation and scheduling, and user directories. The host node may serve multiple users simultaneously even as it spatially partitions the compute nodes among user jobs. The host node may have its own secondary storage, use the clusterwide mass storage, or access an external file system. Users usually log in to the host node through an institution's local area network (LAN).
- Secondary storage—associated with the commodity cluster provides persistent storage for user files and directories, user programs, input data, and result data associated with the jobs that run on the cluster. Logically, the storage is exported to the user through the operating system file system. Physically the storage is a set of disk drives (and possibly tape drives) combined with controller hardware and connections between the controllers and the hard drives. Each node may have its own disk(s) built into the node or have access to a set of disk drives comprising the systemwide file system, sometimes referred to as the "storage area network". Alternatively, the file system may be external to a cluster in the form of a network file system.

The canonical commodity cluster block diagram shown in Fig. 3.1 contains these principal components.

3.1.4 IMPACT ON TOP 500 LIST

As previously discussed, the field of HPC tracks its progress by measuring the performance of systems executing a derivative of the Linpack benchmark, high performance Linpack, and listing the fastest 500

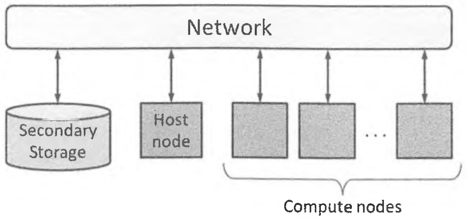


FIGURE 3.1

Commodity cluster components.

Scalability—unlike symmetric memory processor (SMP) systems, commodity clusters are scalable in that the number of nodes can vary widely dependent on need, space, power, and cost. As above, for a given scale the cost is probably less than for alternative custom systems. However, this claim may be primarily in the domain of throughput computing rather than capability computing, for which the overheads, latencies, and bandwidths may be less substantial than using custom systems.

Configurability—the flexibility of configuration for commodity clusters has been historically greater than vendor-configured custom systems. Not only is variability of scale more flexible, but topology of node interconnects, node memory, and processor sockets, external input/output (I/O) components, and other properties can be easily specified by the end-user institution. Because of exploitation of industry standards and multiple sources of system elements, a diversity of choices gives even greater flexibility and more alternatives. Also, systems can be modified over time rather than remaining stagnant throughout their lifetime.

Latest technology—clusters are made of subsystems that have large markets, hence the economy of scale through mass production. As a consequence, such subsystems are targets by vendors for incorporation of the latest technologies to remain competitive in large markets such as enterprise servers or SMP platforms. The integration of these leading-edge subsystems guarantees that commodity clusters, even those provided by system integrators, will incorporate the state of the art in component technologies.

Programming compatibility—while very different in appearance and cost, commodity clusters are compliant in for if not function to massively parallel processors (MPPs). The approach to programming both is quite similar although optimizations may differ. Both clusters and MPPs consist of microprocessor cores, tightly coupled nodes of processors and memories, and integration networks of various topologies. This permits an MPI to be employed in programming both classes of supercomputer, and the use of OpenMP for programming the individual system nodes. This compatibility permits application codes and libraries to be shared between clusters and MPPs, and similar skill sets to be used for both as well.

Empowerment—a sociological aspect of commodity clusters, unanticipated by their original developers, was that users in labs and academia principally found that they had control of their system and were not bound or constrained by commercial fixed product specifications or proprietary software. There was an excitement about supercomputers, an ease of engagement in doing it with off-the-shelf components, and it was fun. Due to this a whole new generation was attracted to this form of supercomputing, and it continues to draw young people into the field to this day.

3.1.3 CLUSTER ELEMENTS

There are widely varying alternative structures for commodity clusters, this being one of their features for scalability and configurability. But in one form or another, almost all clusters comprise the same four classes of component types.

- Node—the principal element that contains the major processing and main memory components to perform user computations. The node is a standalone computer capable of handling independent user workloads. Even as part of a larger cluster, a node may be used to perform a single computation with other nodes doing separate work in the mode of throughput computing. Additional components of nodes usually include communication channels for processor to machines every 6 months. Over this period, observed performance has experienced a gain factor of more than a billion ($>10^9$ times). Fig. 3.2 represents this history in terms of the class of HPC systems contributing to the list at any period of review. Commodity clusters did not even show up on the list until 1997, with the entry of the network of workstations (NOW) system. In 2005 commodity clusters constituted half of all systems in the Top 500 list; today that proportion has grown to about 85%, and has been above 80% for the last 8 years.

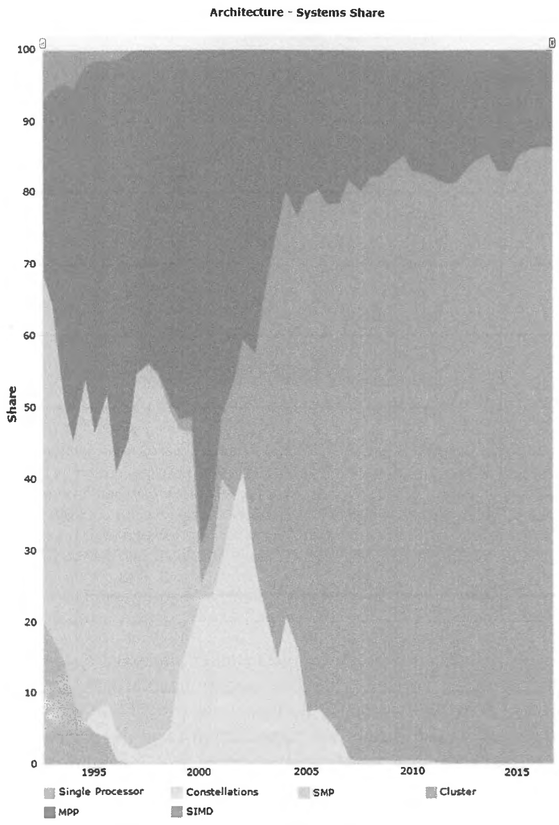


FIGURE 3.2

The dominant system architecture classes comprising the fastest 500 computers over the last 24 years.

Courtesy Top500.org

3.1.5 BRIEF HISTORY

Robert Metcalf. *Photo from wikimedia commons*

Robert Metcalfe is an American electrical engineer who helped to invent Ethernet while working at the Xerox Palo Alto Research Center. He also gave Ethernet its name, borrowing the term from the ether incorrectly thought to be the medium for light propagation in the 19th century. After leaving Xerox he founded 3Com Corporation in his Palo Alto apartment in 1979, initially producing network adapters and going on to produce a wide range of computer network products. 3Com was ultimately acquired by Hewlett-Packard in 2010 for $2.7 billion. Robert Metcalfe is the recipient of the National Medal of Technology, IEEE Medal of Honor, ACM Grace Murray Hopper Award, and IEEE Alexander Graham Bell Medal.

The general strategy of clustering or the implementation of larger systems from smaller fully operational computer systems is not new, and goes back to the 1950s. Among the most prominent system was the IBM SAGE multiple computer developed for the US Air Force NORAD air defense system to acquire and display radar data of incoming aircraft (over the polar regions) to provide an early instance of automated situational awareness and control. SAGE comprised a number of systems that were derivatives of the Whirlwind computer developed by MIT in the late 1940s, a 16-bit vacuum tube architecture that was the supercomputer of its day. The SAGE cluster was motivated by the need for high throughput of large (for that day) datastreams and many-user simultaneous access. It was a pioneer in the use of real-time visual display and screen—input interface. SAGE was also designed to provide enhanced reliability through the multiplicity of identical component systems. When one failed, the others could assume the added workload for nonstop operation.

Although not as exotic, two-way clusters were adopted commercially in the late 1950s, where one system was employed to perform the computing workload while the other was assigned the task of controlling I/O devices such as tapes, disk drives, punched-card handlers, and printers. The IBM 7090/7040 was a very successful example of these dual-node commercial clusters.

The term "cluster" itself was first employed in the late 1980s by Digital Equipment Corporation for its Andromeda project. This early cluster combined 32 VAX 11/750 minicomputers and served as a testbed for experimental studies of hardware system interconnection and software support. This cluster system was never commercialized.

During the 1970s and 1980s network technologies were devised to serve as LANs to allow multiple standalone computer systems in the same environment to share resources such as file systems, laser printers, and ports to external wide area networks such as the emerging internet. The IBM token ring, ATM, and Ethernet were among the most prominent of these. Ultimately, Ethernet emerged as dominant and eventually superseded the others. Ethernet, developed by Metcalf and Boggs [4], was first successfully deployed commercially as a multidrop carrier-sensed arbitration protocol in 1980, permitting multiple systems to share a single interconnection framework to move data between them. These first Ethernet LANs operated at a peak bandwidth of 2.94 Megabit/second in 1973, but were soon replaced with a more pervasive 10 Megabit/s commercial standard in the mid-1980s. It was the establishment of dependable and relatively low-cost LAN interconnection capabilities that set the stage for the next steps to commodity clusters.

The microprocessor was invented by Intel in 1971 [5] and by 1980 was employed in a number of personal computers (PCs), such as the TRS 80, Apple 2, and IBM PC. These were quickly followed by higher grades of single-person direct-access computers employing 16-bit microprocessors referred to as workstations, including the IBM RS-6000, the Sun-1, and others. PCs offered a consumer-grade computer that was relatively slow, with limited main memory and storage, and limited resolution and screens. Their saving virtue was their price, due to large and growing mass market resulting in economy of scale. Workstations provided far more performance, memory, storage, and visual presentation resolution for industrial-grade applications. The difference in cost between a workstation and a PC could be as much as an order of magnitude.

In the late 1980s and early 1990s the combination of LANs integrating workstations offered an opportunity that emerged as workstation farms for sharing batch workloads made up of many user jobs. It was recognized that workstations were often high-availability, low-utilization devices supporting real-time user access but not always fully busy with executing intensive computing tasks. Software was developed, such as the widely used Condor system by Miron Livny, to distribute pending jobs from workstations within the farm to idle workstations. At the job grain boundary, such farms supported parallel job throughput or capacity computing. This would portend the later evolution of workstation clusters.

In 1993 Chuck Sites, former professor at California Institute of Technology, invented the SAN, optimized to connect workstations so they could work together on a common workload. The Myrinet, manufactured and marketed by Myricom, had significantly lower communication latency and higher bandwidth than prior LANs.

Also in 1993 two cluster projects were started: the UC Berkeley Network of Workstations (NOW) Project and the NASA Beowulf Project. The NOW Project pursued the premise that many powerful small-scale computers, in this case workstations, could together outperform a single very large computer such as a mainframe or supercomputer of the day. The NOW Project implemented a series of clusters constructed from high-end workstations, specifically SUN workstations, and the Myrinet network. By 1997 the NOW Project was represented as the first cluster on the Top 500 list; the same year as the first Teraflops computer, the Intel ASCI Red.

As discussed in the next section, the NASA Beowulf Project used a distinctly different approach, incorporating low-end consumer-grade PCs and integrating them with the widely used Ethernet LAN. It also introduced Linux to the supercomputing community. For much of the succeeding 2 decades, this formula dominated commodity clusters and ultimately supercomputing as a whole.

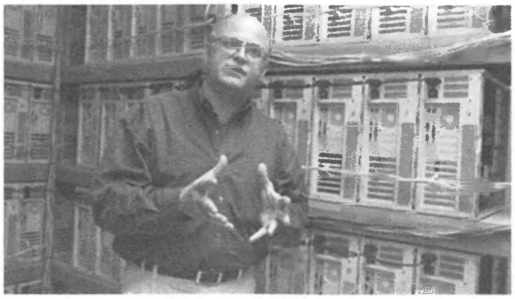
Throughout the late 1980s and the early 1990s a series of message-passing programming interfaces were developed by industry, national laboratories, and academia. These represented in one form or the other the communicating sequential processes execution model derived by Anthony Hoare in the late 1970s [6]. Ultimately this body of work culminated in the communitywide application programming interface being developed and agreed upon, MPI. Shortly thereafter, MPI over CHameleon (MPICH) [7] was developed as a first reduction to practice by Argonne National Laboratory, making MPI widely accessible to a broad user community and establishing it as the premier programming library for both MPPs and commodity clusters.

By the late 1990s commodity clusters emerged as one of several forms of supercomputing systems contending for supremacy. By 2005 and throughout the succeeding decade commodity clusters achieved primary status in terms of number of deployed systems on the Top 500 list.

3.1.6 CHAPTER GUIDE

This chapter describes commodity clusters in breadth, providing an overview of this class of supercomputer and a good picture of supercomputing overall. The next section delivers a technical history of a major milestone in the development of commodity clusters, the Beowulf Project in the mid-1990s, and in so doing introduces many of the components, hardware and software, making up the modern commodity cluster. Section 3.3 gives a detailed description of the hardware components making up the commodity cluster, including the processing nodes, the SAN for integration, communication, and synchronization, and the mass storage for nonvolatile data archiving. Section 3.4 presents the principal ways in which commodity clusters are programmed, involving sequential programming languages combined with parallel programming interfaces and libraries. Section 3.5 provides a software counterpart to the previous hardware discussion, describing the principal system software components, environments, and tools such as the operating system and resource management middleware. Section 3.6 begins the process of hands-on skill-set development. It walks the student through the simplest of actions for logging in to clusters and moving through the user directory hierarchy, running parallel programs, compiling application source codes, and visualizing result data, among other tasks. Section 3.7 closes with a set of conclusions and outcomes of the chapter content.

3.2 BEOWULF CLUSTER PROJECT


Thomas Sterling in front of a commodity cluster built as part of the Beowulf Project. Such commodity clusters are now frequently referred to as belonging to the Beowulf class of supercomputer.

Thomas Sterling is widely known as the father of the Beowulf class of supercomputers. His pioneering work in 1994 along with Don Becker in creating a cluster comprised of commodity-grade computers, collectively referred to as a "Beowulf cluster", significantly reduced the cost of supercomputing and later resulted in the widespread adoption of commodity clusters for scientific computing. This effort resulted in Beowulf being awarded the 1997 Gordon Bell prize in the price—performance category. The Beowulf Project's adoption and software support of the Linux operating system also contributed to the widespread adoption of this operating system in supercomputing systems worldwide. Apart from being the "father of Beowulf", Thomas Sterling's contributions to the hybrid technology multithreaded architecture based on superconducting logic continue to have impacts on high-end computer system architecture design. Thomas Sterling is the recipient of the American Association for the Advancement of Science and HPC Vanguard Awards.

As the history of cluster computing in the previous section indicates, many projects from industry, academia, and government across the international community contributed to the culmination of commodity clusters as the dominant form of supercomputing applicable to a wide range of problem domains and system scales. Nonetheless, one project stands out as the preeminent milestone in the emergence of commodity clusters into the mainstream: the Beowulf Project, begun in the fall of 1993 at the NASA Goddard Space Flight Center by Thomas Sterling and James Fischer, and soon joined by Donald Becker at the start of 1994.

The Beowulf Project explored the potential of deploying systems of a peak performance of 1 Gigaflops and sufficient memory and disk storage to hold scientific data for problems of interest, notably in the earth and space sciences, at a cost that justified systems dedicated to individual computational scientists. At this time, the cost of a high-end workstations was about $50,000 and a

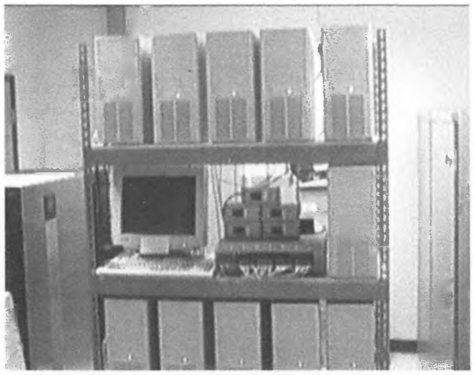


FIGURE 3.3

The 1996 1 Gigaflops Beowulf cluster.

system capable of the abovementioned peak performance was around a million dollars. In the summer of 1994 the first system, "Wiglaf", was deployed, constructed of 16 PC nodes each with one Intel 80486 at 100 MHz, 16 Mbytes of memory, and dual 10BASE-T Ethernet, at a cost of approximately $40,000. By 1996 the third-generation Beowulf Project proof-of-concept system, "Hyglac", achieved sustained performance on a real code in excess of 1 Gigaflops, again with 16 nodes (Fig. 3.3). This time the processors were Intel Pentium Pros at 200 MHz, with a total of 2 Gigabytes of main memory and employing a much-improved 100BASE-TX Fast Ethernet with a nonblocking switch. While the details have changed, this represented the trajectory for future mainstream commodity clusters.

The Beowulf Project also broke ground in its use of the open-source Linux operating system. It made major contributions by providing almost all network driver software employed by Linux at that time and for years to come (for which Donald Becker would rightfully be credited worldwide). This began a process by which Linux ultimately became the number one operating system used in the field of supercomputing up to this day. The Beowulf Project adopted the use of the initial MPICH libraries developed by Bill Gropp and his team at Argonne National Laboratory, which too through evolution would become the standard for programming distributed clusters.

By 1997 a new phase of the Beowulf Project engaged multiple research sites. That year a joint team including Salmon, Warren, Becker, and Sterling won the Gordon Bell Prize for performance to cost. This team also presented a series of tutorials at various conferences that ultimately led to the publication of the popular "How to Build a Beowulf" by Sterling, Salmon, Becker, and Savarese by MIT Press [8]. Later, a second, more comprehensive and up-to-date book, also published by MIT Press and entitled "Beowulf Computing with Linux", was authored by Bill Gropp, Rusty Lusk, and Sterling, and eventually went into a second edition [9].

Beowulf is not the creation of a single individual or even a group; rather it is a synthesis of a set of diverse accomplishments in hardware technology, software libraries, and application programming developed concurrently by many different individual contributors, teams of experts, and product vendors. One example is the community driven development of the MPI programming interface and the MPICH reduction to practice. But while considered an obvious outcome, it was not obvious to mainstream practitioners at the time—in fact for at least 3 years there was strong resistance, even intransigence, at all levels of the supercomputing community to the introduction of commodity clusters as a medium of HPC. It was the vision, tenacity, and trial and error or experimentalism of the early explorers that brought this strategy in hardware and software to the realm of scientific and engineering problem solving. Eventually, the vendors themselves recognized the market opportunity and enhanced aspects of node systems and packaging as well as networking to facilitate commodity clusters and their effective usage. Many hardware vendors and independent software vendors would provide an ever-growing customer base with higher-density, full-featured, highly scalable, and more efficient commodity cluster systems.

Today both within the United States and worldwide, commodity clusters have become a vehicle to excite and educate college students in parallel processing and supercomputing. Major contests are held every year at both the Supercomputing Conference in the United States and the International Supercomputing Conference in Germany. Even high-school students have been attracted to the hands-on aspects of Beowulf computing. In closing, the term "Beowulf computing" was not coined by the original team of developers; they were only responsible for naming a project Beowulf and so injecting the word into the lexicon. Someone else in the public media, and it may never be known whom, used the phrase in print and it caught on. More than 20 years after its first humble and uncertain beginnings, Beowulf computing in the form of commodity clusters now dominates the field of supercomputing.

3.3 HARDWARE ARCHITECTURE

Commodity cluster hardware is, by definition, all commodity off the shelf (COTS) to maximize the benefit of economy of scale and achieve the best performance to cost. The hardware architecture of the commodity cluster is therefore driven and constrained by this requirement. As briefly discussed in Section 3.1, the principal system components in a commodity cluster are the computer nodes, SAN, host node, and mass storage. The architecture of a cluster exploits these resource classes, but is also limited by them and additional support components (e.g., graphics processing units) that conform to industry interface standards. The hardware architecture of a cluster reflects the choices of the specific component types, their number, and the structure in which they are organized and integrated via associated networks. These components and their effect on system architecture are described below, with further expansion and details on specific component types presented in later chapters.

3.3.1 THE NODE

The principal system component of the commodity cluster is most commonly referred to as the "node", and includes most of the active components that make up the aggregate cluster computer. The replicated nodes, also referred to as "the compute nodes", in combination with the integrating interconnection network and the mass (secondary) storage comprise the complete scalable commodity cluster with associated mass storage. But the node itself is a full and self-contained computer that alone and individually serves a much larger user market and therefore benefits from economy of scale to deliver exceptional performance relative to cost, at least for some important institutional workloads. The peak performance and capacity of a cluster are essentially the aggregate capability of all of the compute nodes in combination.

The responsibility of the node is to perform useful computing work for the end user. This is achieved through the collection of processor cores. A modern cluster node is made up of one or more multicore chips, also referred to as "processors", "sockets", "processor sockets", and so on. Depending on the type and number of cores on the chip, the term "many core" may be used. A core is the workhorse of any modern computer. It issues a sequence of user instructions, each potentially designating multiple operations to be performed. A multistage execution pipeline carries out the microoperations required in succession to complete a given instruction and retire it when results are written back into the associated register, to the memory system, or to an I/O channel (other effects are possible as well). A processor socket contains multiple cores, one or more layers of memory cache for high memory bandwidth and low latency access, and chip networks that integrate the cores, caches, and external I/O ports together. In many cases a cluster node incorporates multiple multicore processor sockets. As is discussed in more detail in the following chapters, the microarchitecture of the node cores varies depending on manufacturer and system integrator. Popular processors have variants of two separate architectures by Intel, the IBM Power architecture family, x86 variants by AMD, and the ARM architecture, which is becoming increasingly interesting although it has not as yet had significant impact on the cluster market.

The node is also the container for the main system memory, which primarily uses DRAM technology. Although there are many variants of technology and design, a typical DRAM bit cell consists of a switching transistor and associated capacitor for high-density, low-cost, and moderate-speed data access, both read and write. Although core and main memory are both made from semiconductor devices, they are usually on separate chips because the respective manufacturing processes are very different for optimal behavior of each. Multiple DRAM chips are mounted on single cards, and a number of cards are plugged into industry standard interfaces. The sum of these cards determines the total main memory capacity of the node, with the number of nodes then determining the total main memory of the commodity cluster.

The onboard network channels of the node support intranode communication to move data between the processor sockets, the main memory boards, and the external I/O ports of the node. These networks are transparent to the user and controlled either by the low-level "chip-set" also on the node motherboard or by the node operating system. One of these communications channels is open to the user institution at the time of deployment or when reconfiguration is being conducted. The PCI "bus" is a standardized multiport I/O device that permits additional subsystems to be added in a "plug-and-play" manner to the node without additional hardware changes. Several generations of PCI interconnects have been employed in succession, the latest being PCIe. Even for this single specification, there are many distinct scales for each generation. Other interface ports, some of which go through the PCI bus on the node, are available, such as the ubiquitous USB ports and more obscure accesses for maintenance and administration. Of particular importance is possible direct access to hard disk drives for secondary storage, network controllers for the LAN, and an additional NIC for the SAN discussed below.

3.3.2 SYSTEM AREA NETWORKS

The SAN is the central and differentiating attribute of a commodity cluster that varies in industry standards for communications. It is the principal distinction between a commodity cluster and a more generalized clustering of components where the network is custom designed, such as the Intel Omnipath. Many different networks have served this purpose. In 1994–1995 two approaches were explored. The first was the invention of the SAN by Chuck Seitz, a former professor at Caltech, who created and manufactured the "Myrinet" that was very high performance and low latency for its day. It was also expensive. It was employed by the UC Berkeley NOW project that used Sun Microsystems workstations (hence network of workstations), which were also relatively expensive.

The second was the adoption of the Ethernet LAN to this purpose by the NASA Beowulf Project using low-cost, but low-performance, PCs based on the x86 Intel microprocessor architecture. Both approaches were heavily used throughout the following decade in commodity clusters. Myricom, the vendor and distributor for Myrinet, is no more, but Ethernet continues to this day, only recently being surpassed by the Infiniband network architecture, "IBA". Ethernet has dominated the low-cost SAN market and those clusters employed primarily for throughput computing, while IBA is widely used for more tightly coupled commodity clusters with higher bandwidth and significantly lower latency. Both branches of the SAN technology continue to evolve. Some commodity clusters will incorporate both types of network, with the actual computing being conducted over the IBA network and the "out-of-band" activities for administration and system maintenance being performed over the Ethernet network.

SANs comprise physical channels for data transfer over distances of a few centimeters to hundreds of meters. These may be either conductors, usually copper, or optical fiber depending on issues of cost, energy, and bandwidth requirements. They are connected to nodes by NICs. These may be hardwired into the node motherboard, as is found with GigE (1 Gigabit per second Ethernet) in many cases, or with separate NIC cards often plugged into the nodes' PCIe connectors. The NIC converts data provided by the processors or directly from main memory into message packets of varying length to be sent to destination nodes. The third component is the router or switch used to create topologies of multilayer network structures for higher degree of nodes. Switches are characterized by their degree (number of ports) and their time to transfer a packet from input port to output port, including the time to set up the internal switching configuration. For very large systems, switches can make up a large investment and a major part of the system total cost as well as energy usage.

3.3.3 SECONDARY STORAGE

Persistent storage in one or more forms is essential for computing to retain indefinitely user programs, libraries, and input and result data. Commodity clusters may directly employ hard disk drives or solid-state devices (SSDs) built in to each node. Alternatively, a storage subsystem with its own controllers and possibly its own network may be included as a separate unit within the cluster. Finally, the commodity cluster may access an external mass storage system via the LAN and share it with other user systems. A mix of these is possible and often used. One advantage of not having hard disks integrated within the node is a significant reduction in node power consumption and improved reliability. Disk drives are mechanical and therefore have a higher failure rate (like fans), so avoiding them in the node improves the node's downtime. These would be referred to appropriately as "diskless nodes". However, the recent rapid growth of use of nonvolatile random access memories fabricated from semiconductors eliminates the use of mechanicals and can largely resolve this problem. Separate file systems are usually built from faster hard disks and incorporate redundancy (e.g., RAID) to circumvent downtime due to single disk failures. This is often a cost-effective and operationally better approach to providing persistent storage for users.

3.3.4 COMMERCIAL SYSTEMS SUMMARY

Table 3.1 gives an overview of several commercially available commodity clusters.

Table 3.1 Overview of Components of Several Commercially Available Commodity Clusters

Machine	Network	Processor	Cores per Node	Memory Capacity	Blades	Vendor	Secondary Storage	Nodes per Rack
SuperMUC	Infiniband-FDR (41.25 Gb/s) Mellanox	Sandy Bridge–EP Intel Xeon E5-2680 8C, 2.7 GHz (Turbo 3.5 GHz)	16	32 GB/node	Yes	IBM/ Lenovo	15 PB (scratch) 3.5 PB (home)	512
Mistral	Infiniband-FDR	Xeon E5-2680v3 12C 2.5 GHz/ E5-2695v4 18c 2.1 GHz	24	64 GB/node	No	Bull, Atos		18
Cray CS-Storm	Infiniband-FDR	Xeon E5-2660v2 10C 2.2 GHz	20	Up to 1024 GB/node	No	Cray		23
Stampede	Infiniband-FDR (56 Gbps)	Xeon E5-2680 8C 2.7 GHz	16	32 GB/node	No	Dell	14 PB (shared) 1.6 PB (local aggregate)	40
HPC4 HP POD	Infiniband-FDR	Xeon E5-2697v2 12C 2.7 GHz	24		Yes	Hewlett–Packard	1.8 PB (shared) 0.75 PB (midterm shared) 1.5 PB (local aggregate)	160

3.4 PROGRAMMING INTERFACES

The principal programming modes for parallel programming involve using parallel library application programming interfaces that have bindings to sequential languages. This is the main modality that will be presented here.

3.4.1 HIGH PERFORMANCE COMPUTING PROGRAMMING LANGUAGES

Among programming languages frequently used in HPC, the most popular continue to be Fortran, C, and C++. Some other languages are growing in popularity for HPC applications, including the Python scripting language.

Fortran was developed by John Backus at IBM and first released in 1957. The name derives from the original description of the language as a formula translating system, and it is designed to be well suited for high-level programming of numerical calculations. In fact, many have described Fortran as a domain-specific language for mathematics. Subsequent standardizations followed, including Fortran 66, Fortran 77, Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008.

The C language emerged in the late 1960s and early 1970s from Bell Laboratories using work by Dennis Ritchie. It was first standardized in 1989 and has gone through multiple updates, including C95, C99, and C11. In 1978 Brian Kernighan and Dennis Ritchie published one of the most influential books and tutorials on C programming, "The C Programming Language" [10], which continues to influence C programmers today.

The C++ language emerged in the early 1980s, developed by Bjarne Stroustrup. The name arises from the "++" increment operator and indicates the "evolutionary nature of the changes from C" [11]. Just as in the case for the C language, the C++ creator also wrote a highly influential book entitled "The C++ Programming Language", which has gone through multiple editions and continues to influence C++ programmers strongly. The C++ standard continues to evolve, from C++98 to C++03, C++11, and C++14.

3.4.2 PARALLEL PROGRAMMING MODALITIES

There are three main parallel programming modalities present in most clusters today: throughput computing, message passing, and shared-memory multiple-thread applications.

Throughput computing involves efficiently running a large number of jobs that may be either entirely independent of one another or require minimal communication or coordination between them. An example is conducting an application parameter survey where a single application is run with thousands of different input parameters concurrently to explore its parameter space. Throughput computing is covered in greater detail in Chapter 19.

In contrast to throughput computing, a single message-passing application requires a significant amount of communication and coordination within the application to speed up the time to solution. The principal programming model for achieving this speed up is the communicating sequential processes model, as exemplified by the MPI (Chapter 8).

Like message passing, shared-memory multiple-thread applications also focus on speeding up the time to solution for a single application rather than efficiently executing a large number of mostly independent applications, as in throughput computing. However, as the name implies, shared-memory multiple-thread applications are restricted to shared memory as opposed to distributed memory, as in the case of message passing. The shared-memory multiple-thread parallel programming modality is exemplified by the OpenMP programming model (Chapter 7).

3.5 SOFTWARE ENVIRONMENT

The software environment is a critical element of every computer's operational infrastructure. It exposes and manages functionality supported by hardware, provides different access and usage modalities for different users, manages global and local resources, and offers tools to expand the installed software base further. The latter is accomplished through utilities focusing on development, testing, optimization, configuration, performance monitoring and tuning, and trackable incorporation of new software modules into the existing code base. Below is a necessarily brief discussion of common software components composing a cluster's operational environment. A number of usage examples are also provided to help readers who have not been exposed to this class of systems before.

3.5.1 OPERATING SYSTEMS

The operating system (OS) provides the software environment and services necessary to use the computer and execute custom applications. It consists of a *kernel* that manages hardware resources and arbitrates access to them from other software layers, *system libraries* that expose a common set of programming interfaces permitting application writers to communicate with the kernel and underlying physical devices, additional system *services* performed by the background processes, and various administrator and user *utilities* that comprise programs invoked by users of the computer to accomplish specific minor tasks. The reader is likely familiar with OSs commonly found on desktop and laptop computers such as Microsoft Windows or Apple OS X. Traditionally, however, this space on "big-iron" systems was reserved to several variants of the UNIX OS—a proprietary OS developed by Bell Laboratories in 1970. Thus one could find AIX on IBM machines, HP-UX on Hewlett–Packard computers, UNICOS on Cray, IRIX on SGI, and Solaris on Oracle products, in addition to academic equivalents such as Minix and Berkeley Software Distribution (BSD and its subsequent forks, OpenBSD, NetBSD, and FreeBSD). Note that the series of Apple Mac OSs mentioned above are also a derivative of FreeBSD. Another important UNIX-like OS that has been steadily gaining in prominence over that last 2 decades is Linux. Linux is frequently employed as the OS of choice on servers and clusters, although it is also used in a broad range of mobile computing devices, for example providing the core implementation for Android OS. While successful in mobile and enterprise markets, Linux desktop penetration oscillates at around only 1%–2% depending on the statistics.

The development of the Linux kernel was started by Linus Torvalds in 1991. Since then many individual developers and companies have contributed to its source, making it a truly multiplatform product, effectively supporting an impressive number of hardware devices through the available driver pool and execution environments, ranging from small embedded devices to large multiprocessor systems. The Linux kernel is an open-source product licensed under GPLv2.

On most systems the Linux kernel is accompanied by an open-source suite of libraries and utilities primarily contributed by the GNU Project. These tools were developed and refined over the course of several decades in a massive online collaboration that originated in 1983, and include the following among many other entries.

- C library (*glibc*)
- C, C++, Fortran, and a compiler for several other languages (*gcc*)
- Debugger (*gdb*)
- Binary utilities comprising linker, assembler, symbol table tools, simple archive manager, and others (*binutils*)
- Application build system support (*make*, *autoconf*, *automake*, *libtool*)
- Command-line shell (*bash*)
- Core utilities that support low-level operations on file systems and contents of stored files (*coreutils*, *less*, *findutils*, *gawk*, *sed*, *diffutils*)
- Text editors (*emacs*, *vi*, *nano*)
- Email utilities (*mailutils*)
- Terminal emulator (*screen*)
- Archiving and compression tools (*tar*, *gzip*).

While these utilities provide a near-complete basic UNIX-like operating environment, most Linux distributions include additional open-source software packages, many of them released under a GPL license. They enable more flexible process management, bootstrap service configuration, network tools, improved email client and server programs, graphical environments (X Window System, Wayland), and desktop environments (Gnome, KDE) in addition to a plethora of other special-purpose programs. To ensure broad compatibility, most of the software conforms to the IEEE POSIX standards, effectively enabling drop-in replacement for proprietary implementations.

3.5.2 RESOURCE MANAGEMENT

Large computers employ resource management systems to coordinate accesses to multiple execution units, memory allocation, network selection, and persistent storage allocation. The number of users of even a single machine can easily reach several thousands, and each of them may potentially execute multiple applications with different properties, requirements, and run time. Manual management of every aspect of machine resource allocation by the operators is therefore prohibitive. Fortunately, several sophisticated resource management packages have been developed and are in extensive use today to automate the tasks related to distribution of user workloads across the compute nodes and monitoring the progress of their execution. One of the most widespread resource management systems is Slurm. The example commands described below may be directly used on any correctly configured system equipped with Slurm.

The resource management programs encapsulate user workloads in self-contained units, or *jobs*, that specify at the very least the program to be run, its input and output datasets, the number of nodes (or cores) to be used for its execution, and the maximum time the program is expected or required to work. These parameters are encoded in *job scripts*, which are discussed in greater detail in Chapter 5.

The user informs the system of the intended workload and its resource requirements by submitting a job script to the execution queue. This is done using the *sbatch* command:

```
sbatch job_script
```

If the script `job_script` is an existing file, this command will create a new job, append it to the default job queue, and print a confirmation similar to the following:

```
Submitted batch job 12345
```

In this output, "12345" is a unique job number assigned by the system to the job that has just been queued. The user may subsequently refer to this number to examine the job status using the *squeue* command:

```
squeue -j 12345
```

The resulting output contains among other information the name of the queue the job was stored in (PARTITION field), the name of the job (NAME), the submitting user's ID (USER), and the execution status (ST). The example output may look as follows:

```
JOBID PARTITION    NAME   USER ST    TIME NODES NODELIST(REASON)
12345     batch job_scri user03 R    0:13    1 node01
```

In this particular case, the job was submitted to the "batch" queue by user "user03" and is already running (status "R") on one node of the cluster. Other noteworthy status flags include "CA" for canceled jobs, "CD" for completed, "F" for failed, "TO" for timed out, and "PD" for pending. The latter marks jobs that await allocation of resources to avoid conflicts with other jobs that are already executing.

A pending or running job may be at any time canceled using the command:

```
scancel 12345
```

If successful, the command does not print any confirmation after removing the job from the queue or killing the related executing application and releasing the affected nodes. Errors (for example an invalid job number given as the argument) will cause an explicit error message to be printed on the console.

3.5.3 DEBUGGER

A debugger enables the programmer to step through code in execution, place breakpoints in the code, view memory, change variables, and track variables, among other capabilities. One of the most common serial debuggers available is the gnu debugger (gdb). The gdb debugger is a command-line debugger where the user can give a series of commands to set a break point, continue execution, examine a variable, set a watch point, etc. Table 3.2 gives a few of the basic gdb commands.

To debug a code, that code must be compiled with debugging information included. For most compilers this is accomplished by adding the "-g" flag to the list of compiler flags when compiling. An example of this using the gnu debugger to alter the execution flow of a serial dot-product computation is provided in Fig. 3.4.

Debugging a parallel application on a commodity cluster introduces several complications. These are addressed in further detail in Chapter 14. A simple and straightforward way to debug a parallel application on a commodity cluster is to launch a serial (nonparallel) debugger for each process. An example is provided in Fig. 3.5.

3.5.4 PERFORMANCE PROFILING

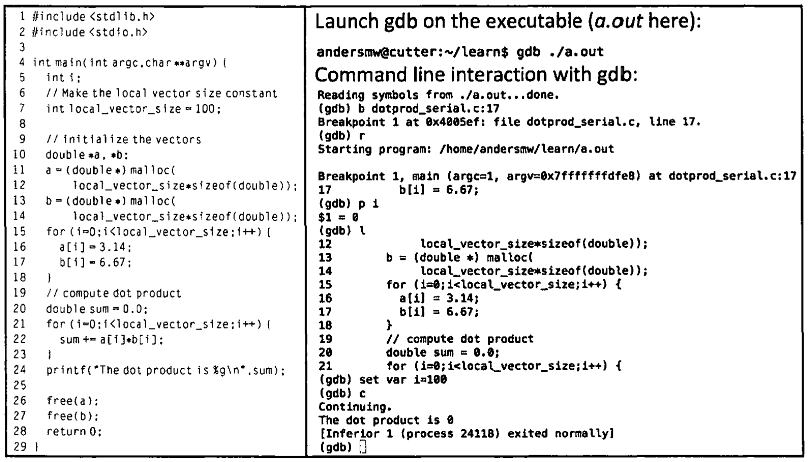
Profiling the performance of an application on a commodity cluster can be carried out in a very similar way to debugging, by launching serial performance profilers on one or several processes. The Linux *perf* utility provides a simple interface for profiling a serial application and can be launched on a single process of a parallel application. An example of launching the serial performance profiler *perf* in conjunction with an MPI supercomputing code is given in Figs. 3.6 and 3.7. Further discussion and details of performance profiling on supercomputers are given in Chapter 13.

3.5.5 VISUALIZATION

There are many open-source and proprietary solutions for visualization of data generated in a commodity cluster. One ubiquitous command-line and script-driven solution is *gnuplot*. An example

Table 3.2 A Few Basic gnu Debugger Commands

gdb Command	Function
gdb <executable name>	Starts the gnu debugger on the specified executable
r	Starts executing the code
l	Lists the current source code where execution is paused
bt	Provides a back trace from the stack
p <variable name>	Prints the variable value
set var <var> = <value>	Sets the value of the specified variable
watch <var>	Sets watch point on specified variables
b <filename>:<line number>	Set break point at specified source code line number
c	Continue execution after pausing after a break point or some other pause
quit	Quit



```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3
 4 int main(int argc,char **argv) {
 5   int i;
 6   // Make the local vector size constant
 7   int local_vector_size = 100;
 8
 9   // initialize the vectors
10   double *a, *b;
11   a = (double *) malloc(
12        local_vector_size*sizeof(double));
13   b = (double *) malloc(
14        local_vector_size*sizeof(double));
15   for (i=0;i<local_vector_size;i++) {
16     a[i] = 3.14;
17     b[i] = 6.67;
18   }
19   // compute dot product
20   double sum = 0.0;
21   for (i=0;i<local_vector_size;i++) {
22     sum += a[i]*b[i];
23   }
24   printf("The dot product is %g\n",sum);
25
26   free(a);
27   free(b);
28   return 0;
29 }
```

Launch gdb on the executable (*a.out* here):

```
andersmw@cutter:~/learn$ gdb ./a.out
```

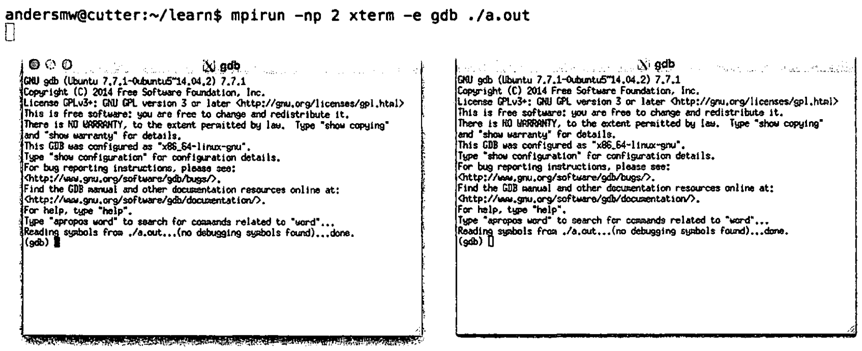
Command line interaction with gdb:

```
Reading symbols from ./a.out...done.
(gdb) b dotprod_serial.c:17
Breakpoint 1 at 0x4005ef: file dotprod_serial.c, line 17.
(gdb) r
Starting program: /home/andersmw/learn/a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffdfe8) at dotprod_serial.c:17
17            b[i] = 6.67;
(gdb) p i
$1 = 0
(gdb) l
12                 local_vector_size*sizeof(double));
13          b = (double *) malloc(
14                 local_vector_size*sizeof(double));
15          for (i=0;i<local_vector_size;i++) {
16            a[i] = 3.14;
17            b[i] = 6.67;
18          }
19          // compute dot product
20          double sum = 0.0;
21          for (i=0;i<local_vector_size;i++) {
(gdb) set var i=100
(gdb) c
Continuing.
The dot product is 0
[Inferior 1 (process 24118) exited normally]
(gdb) 
```

FIGURE 3.4

Example usage of the gnu debugger. The left panel illustrates a simple serial dot-product computation. The right panel illustrates command-line interaction with the gnu debugger, including the setting of break points, printing variables, setting variables, and continuing execution. In the gnu debugger interaction, the loop variable in source code line 15 is reset to be 100, forcing the exit of the loop and the null dot-product result.



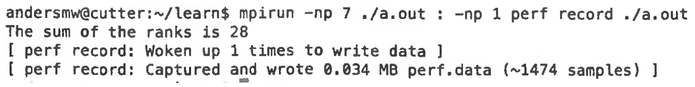
```
andersmw@cutter:~/learn$ mpirun -np 2 xterm -e gdb ./a.out
```

```
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) 
```

```
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 3.5

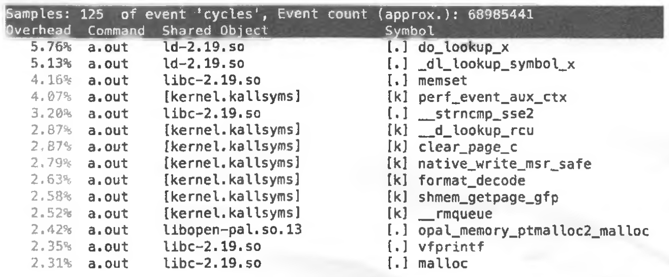
Using the "mpirun" command, the gnu debugger is launched for each process to enable parallel debugging. Two processes are launched here. More details on parallel debugging are given in Chapter 14.



```
andersmw@cutter:~/learn$ mpirun -np 7 ./a.out : -np 1 perf record ./a.out
The sum of the ranks is 28
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.034 MB perf.data (~1474 samples) ]
```

FIGURE 3.6

In this example, a parallel application is run on eight cores where the serial Linux performance counter tool, perf, is run on the application (called a.out here) in just one of those cores. The perf utility is given the instruction to "record" the events in this example for postprocessing. Postprocessing is shown in Fig. 3.7.



```
Samples: 125  of event 'cycles', Event count (approx.): 68985441
Overhead  Command  Shared Object              Symbol
   5.76%  a.out    ld-2.19.so                 [.] do_lookup_x
   5.13%  a.out    ld-2.19.so                 [.] _dl_lookup_symbol_x
   4.16%  a.out    libc-2.19.so               [.] memset
   4.07%  a.out    [kernel.kallsyms]          [k] perf_event_aux_ctx
   3.20%  a.out    libc-2.19.so               [.] __strncmp_sse2
   2.87%  a.out    [kernel.kallsyms]          [k] __d_lookup_rcu
   2.87%  a.out    [kernel.kallsyms]          [k] clear_page_c
   2.79%  a.out    [kernel.kallsyms]          [k] native_write_msr_safe
   2.63%  a.out    [kernel.kallsyms]          [k] format_decode
   2.58%  a.out    [kernel.kallsyms]          [k] shmem_getpage_gfp
   2.52%  a.out    [kernel.kallsyms]          [k] __rmqueue
   2.42%  a.out    libopen-pal.so.13          [.] opal_memory_ptmalloc2_malloc
   2.35%  a.out    libc-2.19.so               [.] vfprintf
   2.31%  a.out    libc-2.19.so               [.] malloc
```

FIGURE 3.7

The postprocessing of the results from the serial performance record in Fig. 3.3 originating from a parallel execution is reported using the "perf report" command.

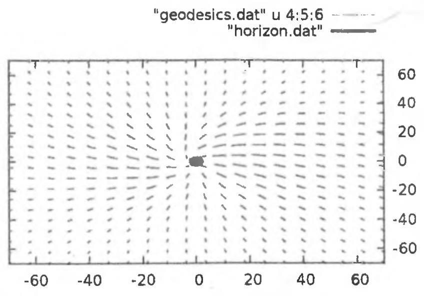


FIGURE 3.8

Visualization of the bending of light around a spinning black hole using gnuplot.

```
#To run this file, type: gnuplot -persist plot.gnu
sp[-70:70][-70:70][-50:50] "geodesics.dat" u 4:5:6 w lines
unset ztics
set style line 1 lt 1 lw 3 pt 3 linecolor rgb "blue"
replot "horizon.dat" with lines ls 1
# bottom
set view 0,0
replot
```

FIGURE 3.9

Script to generate Fig. 3.8. The fourth, fifth, and sixth columns of the file "geodesics.dat" are used to plot the light rays; the spinning black hole horizon is added to the figure from the "horizon.dat" file. These data files are available for download at the textbook website.

of *gnuplot* visualization is provided in Figs. 3.8 and 3.9. Visualization is covered in greater detail in Chapter 12.

3.6 BASIC METHODS OF USE

3.6.1 LOGGING ON

Some readers may already be familiar with the login procedure on common desktop machines. It typically requires providing the user identifier (or clicking on a corresponding icon) and typing a correct password to verify that the person attempting to log on is the same as the one who set up the password. Unfortunately, this method requires direct proximity to the target computer, which is not practical with systems hosting thousands of users or located far away. Hence the login has to be performed over the network utilizing a computer local to the user to act as a connection client. Most supercomputers provide the Secure Shell (SSH) managed logins, which require that an SSH client is installed on the user's machine. Secure connections are preferred, as they thwart most eavesdropping attempts by using strong encryption of all communications, including login information and passwords. Most UNIX-like computers are typically configured to include the SSH program used for this purpose, and on Windows one can install the popular PuTTY package to achieve the same goal. While PuTTY provides a dialog window that manages the login procedure, a login sequence using a command prompt in a terminal window or console is described below. For those unfamiliar with UNIX systems, launching the program called *xterm* is recommended to get the command prompt. Depending on the system, there may be other graphical terminal emulators available, such as *gnome-terminal*, *konsole*, or *urxvt*. If using a console directly (and after a successful login to the local computer), secure communication with a target computer is achieved after typing at the command prompt:

```
ssh -l user03 cluster.hostname.edu
```

In this example the SSH client connects to the account of `user03` on the login node `cluster.hostname.edu`. Note that the user ID refers to the user's login name on the *target* machine, here identified as `cluster.hostname.edu`, and not the one on which SSH was invoked. After the connection is established, the SSH client prints a password prompt on the client machine and the user should supply the account password, again for the target machine. Alternatively, the above command may be invoked as:

```
ssh user03@cluster.hostname.edu
```

If the password is accepted, SSH will respond with the remote machine command prompt in the local terminal. Arbitrary commands may be entered at this prompt and executed in the same way as if they were invoked directly on the remote machine. To finish the interactive session on the remote computer and return to the local shell prompt, it is necessary to type `exit` or simultaneously press Control and D keys.

Another utility, *scp*, is a close companion to SSH and often installed with it. It enables secure transfer of files between the local and remote computers. For example:

```
scp ./myfile user03@cluster.hostname.edu:
```

This copies the local file `myfile` from the current working directory on the local machine to the home directory on the `cluster.hostname.edu` host. Note that the colon following the remote host name is required to inform the *scp* that the second argument is a host and not a file name. Transfer of directories is accomplished by specifying the `-r` option:

```
scp -r user03@cluster.hostname.edu:/tmp/user03/dir
```

This command copies the directory `/tmp/user03/dir` with all its contents to the current working directory on the local machine. An absolute path was specified for the source directory, but only its last component, `dir`, will be created and populated on the local machine.

3.6.2 USER SPACE AND DIRECTORY SYSTEM

Persistent information in a computer has to be stored in a secondary storage system (such as a disk or SSD), since the contents of RAM are volatile. *File systems* organize this information into hierarchical name spaces, where each chunk of data can be properly named and attributed for access. The individual datasets and program executables are stored in *files*. Which parts of the data belong in each file and how to subdivide the computational datasets into files is at the discretion of the user. Each file in a file system has a unique name by which it can be accessed by executing programs. Files are stored in logical containers called *directories*; a directory may also include other directories, permitting building of tree-like structures with arbitrary depth. Like files, directories also have unique names. UNIX-like systems use single-rooted name spaces for all file systems in current use.

A *path* describes a file or directory identifier that is sufficient to locate it in the file system hierarchy. Path components, from left to right, name the subsequent containing directories in descending order (from the root downwards). The last component of a path identifies the target file or directory in its immediate containing unit. In UNIX (and therefore Linux), path components are separated by forward slashes ("/"), hence the individual directory or file names should not contain forward slashes to avoid confusion. By convention, "/" denotes the topmost directory in a file system, referred to as the "root". "/tmp/myfile" identifies a file called "myfile" stored in a directory called "tmp", one level below the root. The typical location of files owned by a specific user on most machines is in the directory "/home/*user*", where "*user*" is substituted with the login ID of that user. The users are free to create their own subtrees of directories and files in these locations. While the exact details of name space taxonomy may vary from system to system, in Linux they are regulated by an informal specification, Filesystem Hierarchy Standard (FHS) [12], maintained by the Linux Foundation. It defines the typical layout as containing the following directories.

- "/bin" contains critical executables that may be used during system boot
- "/sbin" contains critical system executables that may be used during system boot
- "/lib" includes libraries for the essential executables in "/bin" and "/sbin"
- "/usr" is a root of secondary hierarchy containing mostly read-only data. Notable subdirectories of "/usr" include the following.
  - "/usr/bin" contains nonessential executables, typically system-wide application binaries.
  - "/usr/sbin" contains nonessential system executables such as auxiliary services and daemons.
  - "/usr/lib" holds libraries used by executables in "/usr/bin" and "/usr/sbin".
  - "/usr/include" contains "include" files (headers) used by compilers.
  - "/usr/share" includes shared, architecture-independent data frequently associated with the installed systemwide applications.
  - "/usr/local" stores local, host-dependent data. It has additional subdirectories similar to those of "/usr", such as "bin", "include", "lib", and others.
- "/home" hosts individual user subdirectories with their own settings, configuration files, and custom user datasets.
- "/tmp" is a systemwide store for temporary data that is cleaned after every reboot. While on desktop machines its capacity may be limited, on cluster computing nodes with dedicated storage it is often configured to provide a sizable scratch space with its own data retention policy.
- "/dev" keeps entries representing physical and logical devices under the control of the OS. These entries are not regular files, and special care should be exercised when accessing them.
- "/etc" stores host-specific configuration files.
- "/var" contains logs, spools, email, temporary files, and other variable datasets.
- "/root" provides a dedicated home directory for the superuser (administrator).
- "/opt" is used to store optional packages, frequently third-party proprietary or licensed software.
- "/mnt" contains temporarily mounted file systems.
- "/media" includes mount points for external removable storage media such as USB drives (including flash), e-SATA drives, and CD- or DVD-ROMS.
- "/proc" and "/sys" are pseudo-file systems (not backed up by physical storage devices) providing runtime process data, memory allocation, I/O statistics, performance information, and device configuration and status. These file systems are frequently used by monitoring programs and scripts to obtain access from user space to certain types of information maintained by the OS kernel.

Unlike some other OSs, UNIX-compatible systems do not introduce distinct directory hierarchies for each storage device. Instead, the contents of a file system associated with a specific device are *mounted* under some predefined directory (called a *mount point*). The location of that directory in the hierarchy may be arbitrary, and is typically preselected by the system administrator or a suitable device access daemon. This offers the benefit of being potentially able to reach any of the files and directories available in the entire node by performing a recursive traversal of the hierarchy starting from its root. To examine the currently mounted file systems, the user may issue the "disk free" command:

```
df -h
```

The example output may look as follows:

```
Filesystem               Size  Used Avail Use% Mounted on
udev                      16G   12K   16G   1% /dev
tmpfs                    3.2G  1.7M  3.2G   1% /run
/dev/sda2                235G  146G   78G  66% /
none                     4.0K     0  4.0K   0% /sys/fs/cgroup
none                     5.0M     0  5.0M   0% /run/lock
none                      16G  3.3G   13G  21% /run/shm
none                     100M     0  100M   0% /run/user
/dev/sda1                290M  175M   96M  65% /boot
/dev/mapper/vg0-home     6.8T  4.6T  1.9T  72% /home
```

The devices representing individual component file systems are listed in the leftmost column and the corresponding mount points in the rightmost column. The command also presents the actual size of storage devices suffixed with a proper unit ("K" for kilobytes, "M" for megabytes, "G" for gigabytes, and "T" for terabytes). For users generating a lot of data, "Avail" and "Use%" columns are of particular interest, as they show how much free space is remaining on each device.

To list the contents of any directory an *ls* command is used. If invoked without any options, it will simply print the names of all files in the *current working directory* in several columns. It is far more useful to see various attributes of files, such as their sizes, access permissions, modification time-stamps, and so on. For example:

```
ls -halF /some/path
```

This command is going to output information about all files and directories contained in "/some/path" directory, or, if "/some/path" is a file, only about that file. The output data produced with the "long" ("-l") option shows the file ownership (user and group), its size, the last modification date and time, and file name. Other options add useful features: "-a" will list "hidden" entries (all items whose names start with a period "."); "-h" converts numbers to "human-readable" format with suffixes instead of printing size data in bytes; and "-F" appends a symbol after each name indicating the type of entry. Thus directory entries will end in "/" and executable files in "*". The *ls* command offers many more useful options; to see what they do one can access the relevant manual page by typing

```
man ls
```

Of course, the *man* command can also display information about other commands available in the system. Exploration is strongly encouraged!

To navigate the directory hierarchy, one of the most useful commands is "change directory". For example, typing:

```
cd ..
```

at the prompt will move the shell context to the closest encompassing directory. The double dot notation is a special shortcut to denote the parent directory; similarly, single dot (".") has a special meaning indicating the current directory. This notation introduces another important concept, namely that of *relative* paths. So far, all examples have used paths that begin at the root (the first character is "/"). To reach the final component of such a path, called the *absolute* path, the system needs to start at the root and traverse all component subdirectories. However, it is frequently more convenient (and on occasion faster) to indicate the target location relative to the current working directory. Thus specifying "../tmp/some_file" while located in "/usr" directory would effectively refer to a file described by an equivalent absolute path "/tmp/some_file". To verify that the directory change performed by the last command actually happened, a "print working directory" command may be called:

```
pwd
```

If the previous command was invoked in the user's home directory, the result of the last command will likely be "/home". Another useful path shortcut is a tilde ("~"), which expands to the user's home directory. Hence executing the following will change the working directory to the parent directory of the user's home independent of where it is invoked:

```
cd ~/..
```

Both files and directories can be added to and removed from the hierarchy at will. To create a new directory, a "make directory" command is issued:

```
mkdir /tmp/user13
```

This will create an empty "user13" subdirectory in the systemwide temporary data directory. Since it is owned by the creating user, it may subsequently be used to store arbitrary data attributed to that user. Very few system directories delineated by FHS have this property; typically creation of new entries in system directories by regular users will be denied (for good reasons!) due to insufficient access rights. Note that creating directories with paths containing multiple components that do not exist yet requires a "-p" option (for "parents"). Removal of files and directories is achieved with "remove" or the *rm* command. Thus:

```
rm -r ~/my_jobs
```

deletes the "my_jobs" subdirectory from the user's home. While removing files does not require any special options, for directories one needs to specify "-r" (meaning "recursive") to scan for and eliminate all the contents within the directory. Since the *rm* command is frequently configured with a fail-safe interactive mode that requires the user to confirm deletion of every entry, this is often impractical for subdirectories containing thousands of files. For that reason a "-f" or "force" option may be specified to suppress any confirmation prompts.

Note that *rm –fr* is one the most dangerous commands on UNIX systems. Since there is no undelete functionality integrated with most file systems, all deleted files and directories are usually irretrievably lost.

Another useful set of commands performs moving, renaming, and copying of file system entries. To relocate a file or directory to another location, a "move" command is used, with the first argument being the source path and the second being the destination path:

```
mv /tmp/user13/src ~/dst
```

Interestingly, the outcome of this command depends on whether "dst" exists and is a file or directory. If it is a directory, "src" will be removed from the "/tmp/user13" directory and stored in the "dst" directory (this works independently of the original "src" being a file or directory). If both "src" and "dest" are files, "src" will be removed from "/tmp/user13" and stored in the user's home under a new name, "dst". Since this operation also destroys the original contents of the file "dst", *mv* typically will ask the user to confirm the operation. If "src" is a directory but "dst" is a file, the command unconditionally fails (one cannot move a directory into a file). Finally, if there is no object named "dst" in the home directory, the operation removes the original entry from "/tmp/user13" and stores it in the user's home renamed as "dst". Since no preexisting files are overwritten, *mv* does not issue confirmation prompts in this case. As can be seen, the *mv* command is quite multifaceted in that it combines the semantics of relocating the objects in the file system hierarchy, object deletion, and object name modification.

Most of the remarks described for move can be applied for the copy command, or *cp*. There are two crucial differences: *cp* does not remove the original entry referenced by the source path, and the explicit "-r" option has to be specified for all operations in which the source path is a directory. Thus

```
cp -r ~/data/set5 .
```

replicates "set5" directory in the current working directory, leaving the source directory intact.

While this guide introduces some of the most essential file system operations, the world of POSIX commands has much more to offer. Listed below are other commonly available commands suggested for further exploration that can be carried out by consulting the related manual pages (using the *man* command) in a working system:

- *cat* concatenates the contents of multiple files, but is also useful for printing their contents
- *less* permits browsing the contents of text files, scrolling line by line or page by page, or advancing directly to points requested by the user
- *chmod* changes the access permission flags for a file or directory
- *chown* changes the file ownership (user and group)
- *ln* creates a link (named reference) to a file system object
- *du* computes the total storage usage by a specific file or directory
- *touch* updates the file timestamp or creates an empty file
- *head* prints out the starting lines of a file
- *tail* prints out the final lines in a file
- *wc* computes the count of characters, words, and lines
- *file* guesses the file format based on its contents (not extension)
- *find* searches for specific files and directories
- *grep* searches for patterns and phrases in files
- *uname* prints out brief information about the system in use
- *ps* lists processes in the system
- *top* ranks the processes in order of resource usage
- *kill* sends signals to processes, in particular allowing their termination
- *bash* is the primary shell on most systems.

The suggested topics to master include I/O redirection, pipelines, globing, command aliasing, user environment initialization, variable expansion, job control, and basics of scripting.

3.6.3 PACKAGE CONFIGURATION AND BUILDING

The source code of most software packages is distributed in the form of so-called *tar archives. tar* is a utility to create, examine, and unpack the contents of these archives. They retain the original directory layout and file contents of the package-build directory and include the configuration data required to build the binaries on another platform. In addition, the archives may be compressed to save storage space. The following command lists the contents of the archive "package.tgz":

```
tar tvf package.tgz
```

The archives may use different extensions depending on the used compression algorithm. Thus ".tar" indicates an uncompressed archive, files ending in ".tgz" and ".tar.gz" were prepared using *gzip* compression, while ".tbz2" and ".tar.bz2" are the result of applying the *bzip2* tool. Other compression formats are also possible. Recent versions of *tar* are capable of recognizing the compression algorithm automatically and do not require specific command-line options for that purpose. The archive is unpacked by invoking:

```
tar xf package.tgz
```

Typically the command will create a subdirectory tree in the current working directory containing the package source files and configuration scripts. There may also be README and INSTALL files providing additional configuration and installation instructions. Before initiating the build process, it is worth examining various configuration options to customize the features of created executables and libraries properly. After changing the working directory to the top directory of the unpacked archive contents, it may be done with:

```
./configure --help
```

Most commonly, options like "–prefix" that determines the final installation directory, "–with-mpi" that includes MPI support, and "–with-omp" that enables OpenMP-based multithreading are of interest. The final configuration may be then generated as:

```
./configure --prefix=/home/user13/some_package
```

The last command creates the necessary *makefiles*, which are files that contain various definitions, rules, and commands required to execute the build process successfully. Makefiles are utilized by the *make* utility, which optimizes the build process by only executing commands for which the dependencies are newer than the build target. Default makefile names include "makefile", "Makefile", and "GNUmakefile", although the last should be used only for build scripts that contain GNU-specific extensions. To start the build process, one needs to issue:

```
make -j9
```

While the *make* command alone would suffice, the "-j" option initiates a much faster parallel build using multiple processors in the system. The commonly applied rule of thumb suggests passing it an argument that equals the number of available cores plus one, thus the above command should work well on eight-core platforms. The final step when preparing new packages for use is installation of the generated programs, libraries, and data to the target directory. This is accomplished through:

```
make install
```

3.6.4 COMPILERS AND COMPILING

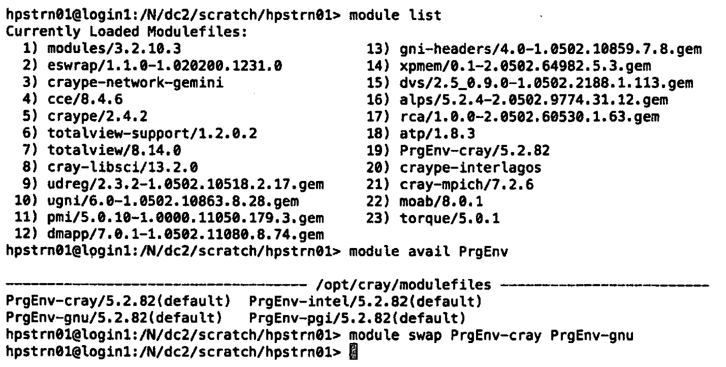
Cluster supercomputers provide several suites of compilers and debugging tools to support the diverse user community using the cluster. The individual user environments are most frequently customized using a module system. Using modules, the compilers and relevant environment paths can be changed in a dynamic way transparent to the user. A list of the most common module commands is given in Table 3.3. An example of module usage on a Cray XE6 is provided in Fig. 3.10.

With the specific compiler flavor and version controlled by loading modules, compiling a source code usually translates into invoking a compiler wrapper and supplying compiler flags along with the

Table 3.3 A List of Commonly Uses Module Commands for Dynamically Controlling the User's Software Environment

Module Command	Description
module load [module name]	Loads the specified module
module unload [module name]	Unloads the specified module
module list	Lists the modules already loaded in the user environment
module avail <string>	Lists the available modules that can be loaded; if a string is provided, only those modules starting with that string are listed
module swap [module 1] [module 2]	Swaps out module 1 for module 2

An example of some usage of these commands is shown in Fig. 3.10. Brackets indicate required arguments, while angle brackets indicate optional arguments.



```
hpstrn01@login1:/N/dc2/scratch/hpstrn01> module list
Currently Loaded Modulefiles:
  1) modules/3.2.10.3                      13) gni-headers/4.0-1.0502.10859.7.8.gem
  2) eswrap/1.1.0-1.020200.1231.0          14) xpmem/0.1-2.0502.64982.5.3.gem
  3) craype-network-gemini                 15) dvs/2.5_0.9.0-1.0502.2188.1.113.gem
  4) cce/8.4.6                             16) alps/5.2.4-2.0502.9774.31.12.gem
  5) craype/2.4.2                          17) rca/1.0.0-2.0502.60530.1.63.gem
  6) totalview-support/1.2.0.2             18) atp/1.8.3
  7) totalview/8.14.0                      19) PrgEnv-cray/5.2.82
  8) cray-libsci/13.2.0                    20) craype-interlagos
  9) udreg/2.3.2-1.0502.10518.2.17.gem     21) cray-mpich/7.2.6
 10) ugni/6.0-1.0502.10863.8.28.gem        22) moab/8.0.1
 11) pmi/5.0.10-1.0000.11050.179.3.gem     23) torque/5.0.1
 12) dmapp/7.0.1-1.0502.11080.8.74.gem
hpstrn01@login1:/N/dc2/scratch/hpstrn01> module avail PrgEnv

----------------------------------- /opt/cray/modulefiles ------------------------
PrgEnv-cray/5.2.82(default)  PrgEnv-intel/5.2.82(default)
PrgEnv-gnu/5.2.82(default)   PrgEnv-pgi/5.2.82(default)
hpstrn01@login1:/N/dc2/scratch/hpstrn01> module swap PrgEnv-cray PrgEnv-gnu
hpstrn01@login1:/N/dc2/scratch/hpstrn01> 
```

FIGURE 3.10

An example of using modules to control a user's software environment dynamically. The first command, module list, lists the modules already loaded. The second command lists the modules available for loading, which begin with the string "PrgEnv". The last command swaps out the Cray programming environment for the GNU programming environment.

source code in the same way as is done when compiling a serial (nonparallel) application. In cluster environments, the C compiler wrapper for applications using the MPI (see Chapter 8) is most frequently called *mpicc*.

3.6.5 RUNNING APPLICATIONS

After accessing compute nodes from the resource management system, as summarized in Section 3.5.2 and detailed in Chapter 5, the user can launch a parallel application on the compute nodes using a shell script to start the computation if using an application in a distributed memory context. In the case of using the MPI, this shell script is most often called *mpirun*. The most important argument it takes is the flag specifying the number of processes to launch, usually given by *—n* <# of processes>. Other options and flags associated with the *mpirun* script can be found by passing to the script the help flag, *-h*. In the case of launching a shared memory application with OpenMP (see Chapter 7), no shell script is needed to start the application.

3.7 SUMMARY AND OUTCOMES OF CHAPTER 3

- A commodity cluster is a group of integrated computer systems. The component computers are standalone, capable of independent operation, and marketed to a much broader consumer base than the scaled clusters which they comprise. The integration network employed is separately developed and marketed to be used by a systems integrator.
- A commodity cluster is constructed from a set of processing nodes, one or more interconnection networks that integrate the nodes, and secondary storage.
- A node of a cluster contains all the components required to serve as a standalone computer.
- Commodity clusters benefit from high performance relative to cost due to the economy of scale achieved by mass production.
- A node incorporates one or more processor cores and sockets, main memory banks, a motherboard controller, an onboard network connecting all the components, external I/O interfaces including an NIC to the SAN, possibly one or more disk drives for nonvolatile storage of data, user program code, and system libraries.
- The principal programming modes for parallel programming involve using parallel library application programming interfaces that have bindings to sequential languages.
- The OS provides the software environment and services necessary to use the computer and execute custom applications. It consists of a *kernel* that manages hardware resources and arbitrates access to them from other software layers, *system libraries* that expose a common set of programming interfaces permitting the application writers to communicate with the kernel and underlying physical devices, additional system *services* performed by the background processes, and various administrator and user *utilities* that comprise programs invoked by users of the computer to accomplish specific minor tasks.
- Large computers employ resource management systems to coordinate accesses to multiple execution units, memory allocation, network selection, and persistent storage allocation.
- A debugger enables the programmer to step through code in execution, place break points in the code, view memory, change variables, and track variables.

- A simple and straightforward way to debug a parallel application on a commodity cluster is to launch a serial (nonparallel) debugger for each process.
- Persistent information in a computer has to be stored in a secondary storage system (such as a disk or SSD), since the contents of RAM are volatile. *File systems* organize this information into hierarchical name spaces, where each chunk of data can be properly named and attributed for access.
- Cluster supercomputers provide several suites of compilers and debugging tools to support the diverse community using the cluster. The individual user environments are most frequently customized using the module system.

3.8 QUESTIONS AND EXERCISES

1. What are the four principal components of a commodity cluster? Describe their functions.
2. Name the required and optional hardware components of a cluster node and describe their properties. Which of them would be more suitable for installation in a compute node and which in a host node? What would be their preferred traits and parameters in each of these environments?
3. Expand and explain the COTS acronym. What is the role of COTS components in a commodity cluster?
4. Contrast a commodity cluster and NOW. What are the drawbacks and benefits of each?
5. List elements of the software environment critical to cluster operation. Which of these components directly involve interaction with the user?
6. What are the steps required to develop and execute a custom application on a cluster?
7. Describe two primary named entities supported by file systems. Why is maintaining consistent organization of file system hierarchy such as the one suggested by FHS important to daily operation of a computing center?

REFERENCES

[1] M. Baker, R. Buyya, Cluster computing: the commodity supercomputer, Software – Practice and Experience 29 (6) (1999) 551–576.
[2] USB-IF, Universal Serial Bus Revision 3.1 Specification, Revision 1.0, July 26, 2013 [Online]. Available: http://www.usb.org/developers/docs/usb_31_061917.zip.
[3] PCI Special Interest Group, PCI-Express Base Specification Revision 3.1a, December 7, 2015 [Online]. Available: http://pcisig.com/specifications/pciexpress/.
[4] R.M. Metcalfe, D.R. Boggs, Ethernet: distributed packet switching for local computer networks, Communications of the ACM 19 (7) (1976) 395–404.
[5] F. Faggin, M.E. Hoff, S. Mazor, M. Shima, The history of the 4004, IEEE Micro 16 (6) (1996) 10–20.
[6] C.A.R. Hoare, Communicating sequential processes, Communications of the ACM 21 (8) (1978) 666–677.
[7] MPICH: High-Performance Portable MPI, [Online], 2017. Available: https://www.mpich.org.
[8] T.L. Sterling, J. Salmon, D.J. Becker, D.F. Savarese, How to Build a Beowulf, MIT Press, 1999.
[9] W. Gropp, E. Lusk, T. Sterling, Beowulf Cluster Computing with Linux, second ed., MIT Press, 2003.
[10] B.W. Kernighan, D.M. Ritchie, The C Programming Language, Prentice Hall, 1978.
[11] B. Stroustrup, The C++ Programming Language, fourth ed., Addison-Wesley, 2013.
[12] The Linux Foundation, Filesystem Hierarchy Standard (FHS), July 19, 2016 [Online]. Available: https://wiki.linuxfoundation.org/lsb/fhs.

CHAPTER

BENCHMARKING 4

CHAPTER OUTLINE

4.1 Introduction 115
4.2 Key Properties of an HPC Benchmark 117
4.3 Standard HPC Community Benchmarks 120
4.4 Highly Parallel Computing Linpack 120
4.5 HPC Challenge Benchmark Suite 123
4.6 High Performance Conjugate Gradients 126
4.7 NAS Parallel Benchmarks 130
4.8 Graph500 132
4.9 Miniapplications as Benchmarks 135
4.10 Summary and Outcomes of Chapter 4 138
4.11 Exercises 139
References 139

4.1 INTRODUCTION

Benchmarking efforts for evaluating the performance of a computer have been ongoing since the beginning of the age of general-purpose computers. The nature of those benchmarks has generally reflected the intended purpose for which the computer was built, while also providing an empirical performance measure that can be compared against the manufacturer's theoretical performance estimate. In the case of the first general-purpose electronic computer, the electronic numerical integrator and computer (ENIAC) (1946) [1], the *de facto* performance benchmark was computing an artillery trajectory and comparing the time to solution against a human computing the same trajectory. Modern supercomputers employ a wide variety of benchmarks, ranging from linear algebra to graph applications, reflecting the diversity of users on modern systems. Just as in the case of the artillery trajectory calculation on the ENIAC, however, user applications are also used on modern supercomputers as *de facto* benchmarks even though they are generally not standardized nor qualified for generic use as a benchmark.

One of the earliest general-purpose benchmarks for evaluating computer performance was the Whetstone [2], named after Whetstone village in Leicestershire, England, where the Whetstone compiler was developed. This benchmark, first released in 1972, consisted of multiple programs that

JACK DONGARRA AND THE LINPACK BENCHMARK

Photo by the University of Tennessee, Knoxville via Wikimedia Commons

Jack Dongarra is one of the most prolific academic researchers, contributing practical advances to high performance computing (HPC) applications, algorithms, and tools over a period of 4 decades. With a principal focus on the central problem of linear algebra critical to many important applications, Dongarra has advanced methods and libraries for effective, efficient, and scalable use of HPC. In so doing he has contributed significantly and provided leadership for such open-source libraries as basic linear algebra subprograms (BLAS), Linpack, Lapack, and ScaLapack among others. From this work came the most widely recognized computing benchmark, highly parallel Linpack (HPL), as a derivative of the earlier Linpack. HPL is the metric by which the Top 500 list of the last 25 years has been measured, essentially defining progress in the field of HPC. Dongarra's more recent work on the high performance conjugate gradients (HPCG) benchmark (conjugate gradient) provided another powerful means to explore the capabilities of emerging HPC systems, stressing more aspects of their architectural properties. Jack is the founding director of the Innovative Computing Laboratory at the University of Tennessee at Knoxville, a Distinguished Research Staff at the Department of Energy Oak Ridge National Laboratory, and a member of the National Academy of Engineering.

created synthetic workloads for evaluating kilo Whetstone instructions per second. In 1980 it was updated to report floating-point operations per second (flops). While this was a serial benchmark not specifically designed for supercomputing systems, it became an industry standard and was used to evaluate the performance of the microprocessors being used in some supercomputers.

In 1984 a benchmark with a standardized synthetic computing workload, named Dhrystone, was released. This benchmark became an industry standard for measuring integer performance. Its name reflects its function as the counterpart to the Whetstone benchmark, but intended for integer performance rather than floating-point performance. Like Whetstone, Dhrystone was not created as a supercomputing benchmark but has been used for evaluating microprocessor components of supercomputers. Dhrystone has since been superseded by the SPECint suite [3].

The genesis of one of the most widely used benchmarks in supercomputing is the Linpack benchmark introduced by Jack Dongarra in 1979 and based on the Linpack linear algebra package developed by Dongarra, Jim Bunch, Cleve Moler, and Gilbert Stewart [4]. While the Linpack linear algebra package has since been superseded by the Lapack library [5] and other competitors, the Linpack benchmark continues to exert a strong influence in the field. It provides an estimate of the system's effective floating-point performance. Beginning in 1979, results from the Linpack benchmark on various systems have been collected by Dongarra. This list started with just 23 computer systems and ultimately grew to include hundreds.

The Linpack benchmark employs a workload that solves a dense system of linear equations. That is, it solves for x in

$$Ax = b \tag{4.1}$$

where b and x are vectors of length n and A is an $n \times n$ matrix with very few or no zero elements. The original Linpack benchmark solved matrices with $n = 100$ and was written for serial computation. No changes to the source code were allowed; only optimizations achieved through compiler flags were permitted. A second iteration of the benchmark used matrices with $n = 1000$ and allowed user modifications to the factorization and solver portions of the code. An accuracy bound on the final solution was also introduced. The third iteration of the benchmark, HPL, allows variations in both problem size and software and can run on a distributed-memory supercomputer. This version of the benchmark is used to generate the Top 500 list that is frequently used to rank supercomputers around the world. Section 4.3.1 discusses HPL in greater detail.

Today there are a wide variety of general-purpose benchmarks used for evaluating the performance of supercomputers and supercomputing elements. These benchmarks often originate from a specific application domain with a workload motivated by that application class rather than from a synthetic workload to achieve better relevance with respect to actual user applications. Table 4.1 provides a brief summary of some of the benchmarks available for HPC users, along with their motivating application domain and characteristics. Some of the most highly used benchmarks come in suites containing multiple individual benchmarks. Two widely used versions are the HPC Challenge suite [6] consisting of seven individual benchmarks (including HPL), and the NAS parallel benchmarks [7] (NPB) consisting of 19 benchmark specifications and reference implementations. These suites are discussed further in Sections 4.5 and 4.7 respectively.

4.2 KEY PROPERTIES OF AN HPC BENCHMARK

HPC benchmarks fulfill several important roles in the HPC community. Benchmarks are frequently used to help decide the size and type of a supercomputer that an institution procures. In this role, many different benchmarks may be used to assess if a candidate supercomputer will adequately address the needs of its users. In a similar role, benchmarks are often called upon to estimate the performance of certain user applications at processor scales and dataset sizes much larger than those available to the user. Benchmarks also help identify and quantify performance upper bounds and limitations for specific application algorithms. For emerging technologies, benchmarks play a key role in comparing performance between conventional practice and a new technology using the same workloads. On many supercomputing systems, benchmarks provide performance milestones against which users can compare their specific application performance and make assessments about the efficiency of their application. Benchmark results form an important historical record for exploring trends in HPC. Finally, benchmarks play an important role in quantifying what percentage of the theoretical peak performance a supercomputer can achieve.

Good benchmarks share several key properties. First, they are relevant and meaningful to the target application domain. Second, they are applicable to a broad spectrum of hardware architectures. Third, they are adopted both by users and vendors and enable comparative evaluation.

Table 4.1 Brief Summary of Some Benchmarks Used in the HPC Community

Benchmark	Application Domain Workload	Aim	Parallelism	Characteristics
HPL	Dense linear algebra	Estimate system's effective flops	MPI (message passing interface)	Part of HPC Challenge suite; used for Top 500 list
STREAM	Synthetic	Estimate sustainable memory bandwidth (GB/s)	None	Part of HPC Challenge suite
RandomAccess	Synthetic	Estimate system's effective rate of integer random updates of memory, reported as giga updates per second (GUPS)	MPI, OpenMP	Part of HPC Challenge suite
HPCG	Sparse linear algebra	Estimate system's effective flops for those applications poorly represented by HPL	MPI + OpenMP	Used for HPCG list ranking
SPEC CPU 2006	Various	Estimate system's effective processor, memory, and compiler performance	None	Commercial
High performance geometric multigrid (HPGMG)	Geometric multigrid	Estimate system's effective evaluation of number of degrees of freedom per second (DOFS)	MPI + OpenMP + CUDA	Used for HPGMG list ranking Comes in two flavors: finite element and finite volume
IS	Computational fluid dynamics	Estimate system's effective integer sort and random access performance	MPI, OpenMP	Part of NPBs
Graph500	Data-intensive applications	Estimate system's effective traversed edges per second (TEPS) for a graph traversal	MPI, OpenMP	Used for Graph500 list ranking

Some other *de facto* properties of successful HPC benchmarks are worth noting. One is that most HPC benchmarks are short. Table 4.2 gives the line count for each of the nonproprietary benchmarks summarized in Table 4.1. While many HPC user applications regularly exceed 100,000 lines of source code, the benchmarks used for evaluating supercomputing resources are generally much smaller.

Table 4.2 Approximate Line Count, Parallelism Application Programming Interface (API), and Language for the Nonproprietary Benchmarks in Table 4.1

Benchmark	Approximate Line Count	Parallelism		Language	
		MPI	OpenMP	C	C++
HPL	26,700	X		X	
STREAM	1500			X	
RandomAccess	5800	X	X	X	
HPCG	5700	X	X		X
IS	1150	X	X	X	
Graph500	1900	X	X	X	
HPGMG	5000	X	X	X	

HPC benchmarks in general specify guidelines about how the benchmark may be run and optimized. Similarly, the results from the benchmark can be archived and shared. Among the benchmarks in Table 4.2, there are four maintained supercomputer ranking lists associated with four benchmarks: HPL [8], HPCG [9], HPGMG [10], and Graph500 [11]. The top supercomputer on each of the lists in June 2017 is provided in Table 4.3.

In addition to providing guidelines for execution, optimization, and result reporting, HPC benchmarks generally use standard parallel programming APIs such as OpenMP and message passing

Table 4.3 The Top-Performing Supercomputer in June 2017 for Each of the Four Benchmarks

Benchmark	Supercomputer	Location	Performance Result	Cores
HPL	Sunway TaihuLight	Jiangsu, China	93.0 petaflops	10,649,600
HPGC	K computer	Kobe, Japan	0.6027 petaflops	705,024
Graph500	K computer	Kobe, Japan	38621.4 GTEPS	705,024
HPGMG	Cori	Berkeley, CA, USA	859 gigaDOFS	632,400

The rank of each supercomputer cross-listed on each list is shown in Table 4.4.

Table 4.4 Ranking of the Top-Performing Supercomputers in June 2017 for Each of the Maintained Ranking Lists

Supercomputer	Top 500 List Ranking	Graph500 List Ranking	HPCG Ranking	HPGMG Ranking
Sunway Taihu Light	1	2	3	2
Tianhe-2	2	8	2	
K computer	8	1	1	
Cori	6		6	1

interface (MPI) (discussed in Chapters 7 and 8, respectively). They also enable the use of different dataset sizes as part of the optimization. For example, the fastest-performing supercomputer for the Graph500 benchmark in June 2016, the K computer in Kobe, Japan, used a graph problem size that was smaller than the third-fastest machine in the list even while they benchmarked against the same type of workload. For HPC benchmarks in general, while the type of workload may be the same within a list of benchmark results, the size of that workload may differ considerably.

One of the most important properties of an HPC benchmark is that its workload should represent some appropriate set of real supercomputer application workloads. This is often one of the most difficult properties for a benchmark, and the performance impact of the type of workload can be significant. The HPCG benchmark mentioned in Tables 4.1—4.4 is intended to complement the HPL benchmark by exploring workloads with data access patterns not exhibited by HPL. The difference between the peak performance of these two types of workloads can be seen in Table 4.3. The fastest HPCG performance is typically less than 1% of the fastest HPL performance, illustrating a huge performance disparity between these two different types of workloads. In June 2017 the notable exception to this was the K computer, which achieves a remarkable 5.3% of the theoretical HPL peak performance. HPC benchmarks with workloads that represent real applications enable better performance estimation and evaluation.

4.3 STANDARD HPC COMMUNITY BENCHMARKS

Sections 4.4—4.8 explore several of the most widely used benchmarks in the HPC community. The most important of these benchmarks, HPL, is part of the HPC Challenge benchmark suite but is singled out in Section 4.4 because of its impact on the HPC industry. The HPC Challenge suite contains seven different benchmarks examining a wide array of memory access patterns and workload types. Complementing the HPC Challenge suite but not part of it is the HPCG benchmark, which covers a large number of applications with workloads not represented by the dense linear solver in HPL and better represents applications with sparse systems of equations. Another important suite of benchmarks is the NPB, which consisted originally of written algorithm specifications for benchmarks with later reference implementations that ultimately ended up becoming the benchmarks themselves in subsequent iterations. This benchmark suite is intended to represent workloads commonly seen in computational fluid dynamics applications. Lastly, the Graph500 benchmark and its associated graph traversal workload are described.

4.4 HIGHLY PARALLEL COMPUTING LINPACK

HPL is one of the most influential HPC benchmarks in the HPC community. It solves a dense system of linear equations and is well suited for floating-point-intensive computations. As noted in Section 4.1, its genesis is the Linpack benchmark introduced by Jack Dongarra in 1979. HPL also serves as the benchmark for determining the supercomputer ranking on the Top 500 list. HPL is written in C and targets distributed-memory computers.

The key workload algorithm in HPL is lower/upper (LU) factorization. Given a problem size n, HPL will perform $O(n^3)$ floating-point operations while only performing $O(n^2)$ memory accesses. Consequently, HPL is not strongly influenced by memory bandwidth and is well suited for empirically exploring the peak floating-point computation capability of a supercomputer.

HPL contains many possible variations in the way it is executed so the best-performing approach for a particular supercomputer can be found empirically. The user is also allowed to replace the LU factorization and solver step reference implementation entirely with an alternative implementation if so desired. Unlike the earlier versions of Linpack, there are no restrictions on problem size in HPL.

HPL is available through netlib.org at www.netlib.org/benchmark/hpl, with the most recent version developed by Antoine Petitet, Clint Whaley, Jack Dongarra, Andy Clear, and Piotr Luszczek. There are two external dependencies for HPL: MPI and the BLAS routines. The compressed tarball is uncompressed as follows:

```
tar -zxf hpl-2.2.tar.gz
```

The directory hpl-2.2 will then appear. In the setup directory there are several examples of compile settings for various architectures. For this example, the make_generic script is executed to produce a template for creating compile settings.

```
cd hpl-2.2/setup
sh make_generic
cp Make.UNKNOWN ../Make.linux; cd ..
```

The Make.linux file now needs to be modified to reflect the location of the hpl-2.2 directory and the BLAS libraries, and the name of the C compiler. In Make.linux the location of the BLAS libraries can be specified in line 97 immediately before the -lblas:

```
95 LAdir    =
96 LAinc    =
97 LAlib    = -lblas
```

A library location is given to the compiler using the –L flag. For example, if the BLAS libraries were located in /usr/local/lib, line 97 would be changed to read:

```
95 LAdir    =
96 LAinc    =
97 LAlib    = -L/usr/local/lib -lblas
```

The location of the hpl-2.2 directory can be specified at line 70. The architecture name can be changed from UNKNOWN to linux in line 64:

```
64 ARCH     = linux
65 #
66 # ----------------------------------------------------------------------
67 # - HPL Directory Structure / HPL library ------------------------------
68 # ----------------------------------------------------------------------
69 #
70 TOPdir  = /your/path/to/hpl-2.2
```

```
0001 HPLinpack benchmark input file
0002 Innovative Computing Laboratory, University of Tennessee
0003 HPL.out      output file name (if any)
0004 6            device out (6=stdout,7=stderr,file)
0005 4            # of problems sizes (N)
0006 29 30 34 35  Ns
0007 4            # of NBs
0008 1 2 3 4      NBs
0009 0            PMAP process mapping (0=Row-,1=Column-major)
0010 3            # of process grids (P x Q)
0011 2 1 4        Ps
0012 2 4 1        Qs
0013 16.0         threshold
0014 3            # of panel fact
0015 0 1 2        PFACTs (0=left, 1=Crout, 2=Right)
0016 2            # of recursive stopping criterium
0017 2 4          NBMINs (>= 1)
0018 1            # of panels in recursion
0019 2            NDIVs
0020 3            # of recursive panel fact.
0021 0 1 2        RFACTs (0=left, 1=Crout, 2=Right)
0022 1            # of broadcast
0023 0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
0024 1            # of lookahead depth
0025 0            DEPTHs (>=0)
0026 2            SWAP (0=bin-exch,1=long,2=mix)
0027 64           swapping threshold
0028 0            L1 in (0=transposed,1=no-transposed) form
0029 0            U  in (0=transposed,1=no-transposed) form
0030 1            Equilibration (0=no,1=yes)
0031 8            memory alignment in double (> 0)
```

FIGURE 4.1

An example parameter file, HPL.dat for HPL. Parameter inputs separated by spaces on each line are each explored and reported independently by HPL to ease the tuning of HPL.

Once the Make.linux file is prepared, HPL can be compiled by issuing the following command:

```
make arch=linux
```

This will create the HPL executable, xhpl, in the bin/linux directory.

Accompanying the *xhpl* executable is a parameter file to tune HPL for the supercomputer. An example is provided in Fig. 4.1.

The parameter space for tuning HPL is very large, so parameter inputs separated by a space on each line are run independently. For example, the default parameter file in Fig. 4.1 will run HPL through 864 ($= 4 \times 4 \times 3 \times 3 \times 2 \times 3$) distinct parameter combinations with a separate report of Gflops for each unique combination.

A brief explanation of the HPL tuning parameters is as follows.

- Lines 1–2 are ignored.
- Line 3 specifies the name of the file where any output should be redirected if requested in line 4.
- Line 4 specifies whether to print the output to screen or to a file.
- Line 5 indicates the number of different problem sizes explored in this parameter file. It cannot be greater than 20.

- Line 6 gives a space-separated list of the matrix problem sizes. If the number of problem sizes given exceeds the number specified in line 5, those excess problem sizes will be ignored.
- Line 7 gives the number of block sizes explored in this parameter file.
- Line 8 gives a space-separated list of those block sizes.
- Line 9 indicates how MPI processes are mapped on to the nodes, and whether row major or column major.
- Line 10 indicates the number of process grid configurations specified in this parameter file.
- Lines 11—12 specify those process grid configurations.
- Line 13 specifies a threshold used for flagging residuals as failed. In general, residuals will be order 1. Specifying a negative threshold turns off checking and allows faster parameter space sweeps.
- Lines 14—31 specify algorithmic variations in HPL. HPL has a number of different algorithm options, including six different virtual panel broadcast topologies (line 23), a bandwidth-reducing swap-broadcast algorithm (line 26), back substitution with a look-ahead depth of one (line 24), and three different LU factorization algorithms (lines 21) among other options. Tuning for these parameters on a specific supercomputer is a routine task with HPL.

  Output for each parameter combination choice is reported in Gflops.

```
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR11C2R4        1000    80     2     2               0.09              7.694e+00
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0072510 ...... PASSED
================================================================================
```

For a certain problem size N_{max}, the cumulative performance in Gflops reaches its maximum value, R_{max}. The R_{max} value is what is reported for the Top 500 list ranking supercomputers. Another interesting metric from the HPL benchmark is $N_{1/2}$, which is the problem size where the maximum performance achieved is $R_{max}/2$.

4.5 HPC CHALLENGE BENCHMARK SUITE

The HPC Challenge benchmark suite consists of seven different tests that cover a range of application types and memory access patterns. The first, HPL, was discussed in Section 4.4 because of its large impact on the HPC community. The other six tests are:

- DGEMM—double-precision matrix—matrix multiplication
- STREAM—synthetic workload to measure sustainable memory bandwidth
- PTRANS— parallel matrix transpose
- RandomAccess—reports the rate of integer random updates of memory in giga updates per second (GUPS)
- FFT—double-precision complex one-dimensional discrete Fourier transform
- B_eff—reports latency and bandwidth for several different communication patterns.

The HPC Challenge benchmark suite can be accessed from the HPC Challenge website [6]. The code is uncompressed and accessed as follows:

```
tar -zxf hpcc-1.5.0.tar.gz
```

The directory hpcc-1.5.0 will then appear. The build methodology for the HPC Challenge benchmark is the same as for HPL: a Make.architecture file is created specifying the compiler and any dependency and optimization information for the supercomputer. Example Make.architecture files are found in the hpcc-1.5.0/hpl/setup directory:

```
cd hpcc-1.5.0/hpl/setup
sh make_generic
cp Make.UNKNOWN ../Make.linux
```

The same changes are made to Make.linux as were done in Section 4.3.1. The benchmark suite is then compiled as follows:

```
make arch=linux
```

This will produce an executable called *hpcc* in the hpcc-1.5.0 directory. The parameter file is called hpccinf.txt and an example version is provided. This parameter file has nearly the same format as the HPL.dat parameter file of Section 4.3.1 but has been augmented slightly to incorporate parameters specific to the matrix transpose benchmark, PTRANS. This change is noted in the parameter file itself:

```
32 ##### This line (no. 32) is ignored (it serves as a separator). ######
33 0                          Number of additional problem sizes for PTRANS
34 1200 10000 30000           values of N
35 0                          number of additional blocking sizes for PTRANS
36 40 9 8 13 13 20 16 32 64   values of NB
```

where lines 1−32 have the same meaning as those in Fig. 4.1. Additionally, the process grid configurations specified in lines 11−12 and the residual threshold in line 13 are used for PTRANS. An example of running the HPC Challenge benchmark is as follows:

```
mpirun -np 16 ./hpcc
```

This produces output from each of the seven benchmarks, summarized in Figs. 4.2−4.8.

```
DGEMM_N=288
StarDGEMM_Gflops=2.44343
SingleDGEMM_Gflops=2.45875
```

FIGURE 4.2

Example DGEMM summary section output.

```
PTRANS_GBs=2.17378
PTRANS_time=0.000628948
PTRANS_residual=0
PTRANS_n=500
PTRANS_nb=80
PTRANS_nprow=2
PTRANS_npcol=2
```

FIGURE 4.3

Example PTRANS summary section output.

```
MPIRandomAccess_GUPs=0.144392
StarRandomAccess_LCG_GUPs=0.11601
SingleRandomAccess_LCG_GUPs=0.118885
StarRandomAccess_GUPs=0.0829133
SingleRandomAccess_GUPs=0.083817
```

FIGURE 4.4

Example RandomAccess summary section output.

```
STREAM_VectorSize=83333
STREAM_Threads=1
StarSTREAM_Copy=5.14952
StarSTREAM_Scale=5.27086
StarSTREAM_Add=7.09093
StarSTREAM_Triad=5.0111
SingleSTREAM_Copy=5.33624
SingleSTREAM_Scale=5.53154
SingleSTREAM_Add=7.25028
SingleSTREAM_Triad=6.75953
```

FIGURE 4.5

Example STREAM summary section output.

```
MaxPingPongLatency_usec=0.55631
RandomlyOrderedRingLatency_usec=0.768096
MinPingPongBandwidth_GBytes=4.28756
NaturallyOrderedRingBandwidth_GBytes=0.533907
RandomlyOrderedRingBandwidth_GBytes=0.576042
MinPingPongLatency_usec=0.238419
AvgPingPongLatency_usec=0.390631
MaxPingPongBandwidth_GBytes=9.36751
AvgPingPongBandwidth_GBytes=6.48206
NaturallyOrderedRingLatency_usec=0.751019
```

FIGURE 4.6

Example b_eff summary section output.

```
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR11C2R4        1000    80     2     2               0.09              7.694e+00
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0072510 ...... PASSED
================================================================================
```

FIGURE 4.7

Example HPL summary section output.

```
FFT_N=32768
StarFFT_Gflops=0.594992
SingleFFT_Gflops=0.613019
MPIFFT_N=262144
MPIFFT_Gflops=6.17472
MPIFFT_maxErr=1.28804e-15
MPIFFT_Procs=16
```

FIGURE 4.8

Example FFT summary section output.

4.6 HIGH PERFORMANCE CONJUGATE GRADIENTS

The HPCG benchmark was created by Jack Dongarra (HPL creator), Michael Heroux, and Piotr Luszczek (HPL developer), with the first release in 2000 and the most recent version released in 2015. It aims to complement the HPL benchmark in exploring memory and data-access patterns in application workloads that are not well represented by HPL. The workload in HPCG centers on a sparse system of linear equations arising from the discretization of a three-dimensional (3D) Laplacian partial differential equation with a 27-point stencil. Like HPL, the workload in HPCG is geared for solving Eq. (4.1), but the A matrix is dominated by zeros in the HPCG workload. Unlike HPL, the solution method in HPCG is driven by a Krylov subspace solver known as conjugate gradient. Krylov subspace solvers are iterative solvers requiring multiple iterations to produce an approximate solution to Eq. (4.1) to within a certain tolerance, and are among the most common methods used for solving sparse linear systems of equations. Because the matrix in HPCG is dominated by zeros, the nonzero elements of the matrix are stored in contiguous memory locations for each row.

The sparse nature of the workload in this benchmark requires many more memory accesses than in HPL. For a problem size n, HPCG will perform $O(n)$ floating-point operations while also requiring $O(n)$ memory accesses. Given this, it is no great surprise that in Table 4.3 the peak flops measured for HPL and HPCG differ by over a factor of 150.

The HPCG benchmark incorporates five major kernels: sparse matrix vector multiplication, symmetric Gauss–Seidel smoothing, global dot product evaluation, vector update, and multigrid preconditioning. In addition, the benchmark provides seven different reference routines, which can be replaced in their entirety by user code optimized for the intended supercomputer in accordance with some specific guidelines.

For Krylov subspace solvers, a significant portion of the solve time is spent in sparse matrix vector multiplication, thereby making the HPCG sparse matrix vector multiplication kernel performance very relevant to performance in many user applications. For a matrix of size $N \times N$ and a vector of size N, matrix vector multiplication is given by Eq. (4.2)

$$x_i = \sum_{j=0}^{N-1} A_{ij} b_j \tag{4.2}$$

where A_{ij} is the (i,j)th element of the matrix and b_j is the jth element of the vector. Because nonzero elements of sparse matrix are stored in contiguous memory locations for each row, Eq. (4.2) can be modified to reflect that zero matrix entries in HPCG are neither stored nor manipulated:

$$x_i = \sum_{j=0}^{n_i} A_{ij} b_j \tag{4.3}$$

where n_i indicates the number of nonzeros in sparse matrix A for the ith row. The sparse matrix vector multiplication kernel in HPCG evaluates Eq. (4.3) in distributed memory requiring some exchange of needed b_j values between memory localities. This is an example of *halo exchange* and is explored in detail in Chapter 9. Both the halo exchange routine and the entire sparse matrix vector multiplication kernel code in HPCG can be replaced or altered with certain restrictions by the user.

Gauss—Seidel smoothing is an iterative solution method for linear systems of equations. The Gauss—Seidel kernel in HPCG tests recursive execution and has memory access characteristics similar to that of the sparse matrix vector multiplication kernel. Like the sparse matrix vector kernel, the entire reference implementation of Gauss—Seidel smoothing can be modified or replaced in the benchmark by the user under specified guidelines.

One of the most important collective communication-type operations in HPCG is computing a global dot product of two vectors in distributed memory to produce a single scalar value available to all processing elements. This type of operation is common in most user applications. HPCG reporting includes the minimum, maximum, and average MPI allreduce time when using MPI for the benchmark. A user-provided dot product routine can be substituted for the reference implementation. The same is true for the vector update and multigrid preconditioner, where all the key kernels of HPCG are tested using four different grid sizes.

Compiling the HPCG benchmark on a supercomputer is straightforward. The MPI-OpenMP reference implementation of the benchmark can be downloaded from the HPCG website, www.hpcg-benchmark.org. The benchmark tarball is unpacked with the following command:

```
tar -zxf hpcg-3.0.tar.gz
```

A resulting directory, hpcg-3.0, containis the source code for the benchmark. HPCG supports out-of-source builds to avoid cluttering the source code directories with build-related files. To compile, a build directory is created; for simplicity this is placed in the hpcg-3.0 directory:

```
cd hpcg-3.0
mkdir build; cd build
```

In the build directory, execute the configure script and pass it the name of the architecture for which the build is desired. The compiler flags and commands for several standard options are already in the setup directory. In the example below, the Linux_MPI configuration option is used.

```
../configure Linux_MPI
make
```

The hpcg executable, called *xhpcg*, will appear in the build/bin directory along with a parameter file named *hpcg.dat.*

Unlike the HPL parameter file, the HPCG parameter file is very simple and only contains four lines:

```
1 HPCG benchmark input file
2 Sandia National Laboratories; University of Tennessee, Knoxville
3 104 104 104
4 60
```

The first two lines are unused and can be replaced with user-motivated descriptions. The third line specifies the dimensions of the problem that are local to each MPI process. Consequently, the global problem size changes for HPCG depending on the number of processes launched, while the local problem size stays the same. In this sense, HPCG is already set up for weak scaling tests. The third line contains three space-separated numbers which correspond to the number of collocation points in a cubic grid used for discretizing the 3D Laplacian partial differential equation that is at the heart of the HPCG workload. The fourth line specifies how long in seconds the benchmark should run (60 s in this example parameter file). To submit official HPCG results, the benchmark should run for at least 1800 s. The parameter file should reside in the same directory as the executable when running.

To run the benchmark using MPI, the mpirun script will launch the executable on the desired number of processes, 16 in this example:

```
mpirun -np 16 ./xhpcg
```

Two files result from this operation, named HPCG-Benchmark-3.0_<today's date and time>.yaml and hpcg_log_<today's date and time>.txt. The hpcg_log file contains the log of output from the HPCG execution; the benchmark results are in the yaml file. Extracts from an example yaml file output from HPCG are shown in Fig. 4.9.

Current HPCG implementations generally achieve only a small fraction of peak flops (most around 1%–2%) on the fastest 10 supercomputers in the Top 500 list while they achieve as much as 90% of the theoretical peak performance for HPL. This highlights the different nature of the HPCG benchmark versus HPL, even though they both report flops as the final overall rating. Fig. 4.10 shows the HPCG and HPL performance for the top 10 supercomputers. One outlier is the K computer, which achieves a staggering 5.3% fraction of peak HPL performance with HPCG. In general, the low efficiency in terms of flops exhibited by HPCG is because even small problems for HPCG do not fit in the (L3) cache, halo exchange and allreduce become network bottlenecks as the number of nodes becomes large, and sparse matrix vector multiplication is limited by memory bandwidth.

```
HPCG-Benchmark version: 3.0
Release date: November 11, 2015
Machine Summary:
  Distributed Processes: 16
  Threads per processes: 1
Global Problem Dimensions:
  Global nx: 416
  Global ny: 208
  Global nz: 208
Processor Dimensions:
  npx: 4
  npy: 2
  npz: 2
Local Domain Dimensions:
  nx: 104
  ny: 104
  nz: 104
```

```
########## V&V Testing Summary ##########:
Spectral Convergence Tests:
  Result: PASSED
  Unpreconditioned:
    Maximum iteration count: 11
    Expected iteration count: 12
  Preconditioned:
    Maximum iteration count: 2
    Expected iteration count: 2
Departure from Symmetry |x'Ay-y'Ax|/(2*||x||*||A||*||y||)/ep
  Result: PASSED
  Departure for SpMV: 3.15835e-08
  Departure for MG: 4.00058e-09

GFLOP/s Summary:
  Raw DDOT: 4.48213
  Raw WAXPBY: 7.70723
  Raw SpMV: 7.3242
  Raw MG: 7.07338
  Raw Total: 7.05082
  Total with convergence overhead: 7.05082
```

```
__________ Final Summary __________:
  HPCG result is VALID with a GFLOP/s rating of: 6.88674
```

FIGURE 4.9

Some extracts from the HPCG benchmark yaml output file. The raw Gflops summary of several of the key HPCG kernels is itemized independently in addition to providing an overall Gflops rating. Verification and validation testing is also reported. The local problem size, 104 × 104 × 104, was specified in the hpcg.dat parameter file, while the HPCG output also reports the global problem size as determined by the number of MPI processes launched.

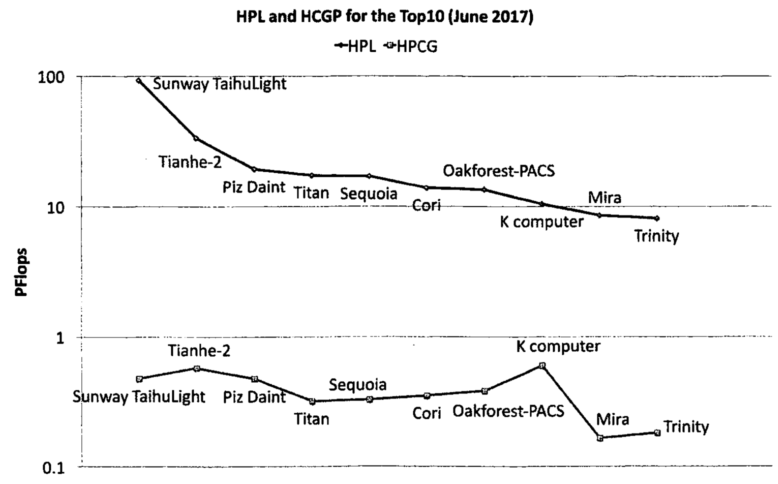


FIGURE 4.10

Comparison of HPL and HPCG performance for the top 10 supercomputers in the Top 500 list in June 2017.

4.7 NAS PARALLEL BENCHMARKS

NPB is a series of small self-contained programs that encapsulate the performance attributes of a large computational fluid dynamics application. It originated from the NASA Ames research center in 1991, and the first version of the benchmark consisted of eight problems that were specified entirely in a "pencil-and-paper" fashion: there was no reference implementation, as in other benchmarks, and the benchmark programs were specified algorithmically. In 1995 the second version of NPB was announced, where reference versions based on MPI and Fortran77 were distributed. Subsequently a third version of NPB was released, which included a number of additions to the original eight problems as additional parallel programming APIs beyond MPI, such as OpenMP, high performance Fortran, and Java.

The original eight problems in NPB are a large integer sort for testing both integer computation speed and network performance, embarrassingly parallel random number generation for integral evaluation, a conjugate gradient approximation to compute the smallest eigenvalue of a sparse symmetric matrix, a multigrid solver for computing a 3D potential, a time integrator of a 3D partial differential equation using fast Fourier transform, a block tridiagonal solver with a 5×5 block size, a pentadiagonal solver, and an LU solver for coupled parabolic/elliptic partial differential equations. These problems are referred to by the two-letter abbreviations IS, EP, CG, MG, FT, BT, SP, and LU respectively. A brief summary of these benchmarks is given in Table 4.5.

The latest version, NPB3, can be downloaded from the NPB page [7] and uncompressed as follows:

```
tar -zxf NPB3.3.1.tar.gz
```

Table 4.5 Some Characteristics of the NAS Parallel Benchmarks (NPB)

NPB	Approximate Line Count	Parallelism		Language	
		MPI	OpenMP	Fortran	C
IS—Integer Sort	1150	X	X		X
EP—Embarrassingly Parallel	400	X	X	X	
CG—Conjugate Gradient	1900	X	X	X	
MG—Multigrid	2600	X	X	X	
FT—Discrete 3D Fast Fourier Transform	2200	X	X	X	
BT—Block Tridiagonal Solver	9200	X	X	X	
SP—Scalar Pentadiagonal Solver	5000	X	X	X	
LU—Lower Upper Gauss–Seidel Solver	6000	X	X	X	

This will create a directory called NPB3.3.1, wherein can be found the MPI versions of the benchmark that are demonstrated here. To compile, enter the NPB MPI version directory:

```
cd NPB3.3.1/NPB3.3-MPI
```

Compiling the benchmark problems requires specifying a compiler choice for C and Fortran in the make.def file located in the config directory:

```
cd config
cp make.def.template make.def
```

Now modify the make.def file in the config directory by specifying the Fortran and C compilers on lines 32 and 78, respectively:

```
29 # ---------------------------------------------------------------------------
30 # This is the fortran compiler used for MPI programs
31 # ---------------------------------------------------------------------------
32 MPIF77 = mpif90

75 # ---------------------------------------------------------------------------
76 # This is the C compiler used for MPI programs
77 # ---------------------------------------------------------------------------
78 MPICC = mpicc
```

Return to the NPB3.3-MPI directory to compile the specific benchmark problem. To compile, three pieces of information must be given to the Makefile: the two-letter (lower-case) reference to the benchmark problem, the number of processes on which to run, and the class of problem where the class is one of S, W, A, B, C, D, or E. S indicates a small problem size; W indicates a problem for a 1990s-era workstation; A, B, and C indicate standard problem sizes increasing by a factor of 4 with each letter; and D and E indicate large test problems increasing by a factor of 16 by each letter.

An example compiling the IS benchmark problem for the smallest problem size on four cores is as follows:

```
cd ..
make is NPROCS=4 CLASS=S
```

The executable will be placed in the bin directory with a name indicating the number of processes and the class for which it was compiled, is.S.4 in this case:

```
cd bin
mpirun -n 4 ./is.S.4
```

Output is shown in Fig. 4.11.

```
NAS Parallel Benchmarks 3.3 -- IS Benchmark

Size:  65536  (class S)
Iterations:   10
Number of processes:     4


IS Benchmark Completed
Class            =                        S
Size             =                    65536
Iterations       =                       10
Time in seconds =                      0.00
Total processes =                         4
Compiled procs   =                        4
Mop/s total      =                   274.91
Mop/s/process    =                    68.73
Operation type   =              keys ranked
Verification     =               SUCCESSFUL
Version          =                    3.3.1
Compile date     =              16 Aug 2016

Compile options:
   MPICC         = mpicc
   CLINK         = $(MPICC)
   CMPI_LIB      = -L/usr/local/lib -lmpi
   CMPI_INC      = -I/usr/local/include
   CFLAGS        = -O
   CLINKFLAGS    = -O


Please send feedbacks and/or the results of this run to:

NPB Development Team
npb@nas.nasa.gov
```

FIGURE 4.11

Output from the parallel IS benchmark for a small class problem size run on four processes.

4.8 GRAPH500

The Graph500 benchmark was announced in 2010, and is intended to represent data-intensive workloads rather than floating-point-intensive computations as in HPL. With support from an international steering committee of over 50 members and led by Richard Murphy, the Graph500 benchmark targets three key problems in the context of data-intensive applications: concurrent search, the single-source shortest path, and the maximal independent set. At present only the concurrent search problem has been specified as Graph500 benchmark 1, and is sometimes known as the Graph500 benchmark. In this subsection the Graph500 benchmark 1 is referred to as the Graph500 search benchmark to avoid confusion.

The Graph500 search benchmark implements the breadth-first search algorithm on a large graph. An illustration of this algorithm is given in Fig. 4.12.

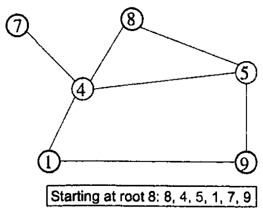


FIGURE 4.12

Example of breadth-first search traversal of this graph data structure starting at vertex 8. The starting vertex is also called the root. The adjacent vertices to the root are 4 and 5, colored red (light gray in print versions). The adjacent vertices to those are 1, 7, and 9, colored blue (dark gray in print versions). Lines connecting the vertices are called edges.

The reference implementation comes with parallelism in several forms, including MPI and OpenMP for distributed and shared-memory settings, and was developed by David Bader, Jonathan Berry, Simon Kahan, Richard Murphy, Jason Riedy, and Jeremiah Willcock. It includes both a graph generator and a breadth-first search implementation. The benchmark starts with a root and finds all reachable vertices from that root; 64 unique roots are checked. There is only one kind of edge and there are no weights between vertices. The output performance metric is traversed edges per second (TEPS). The resulting search tree is validated to ensure it is the correct tree given the root. The graph construction and graph search are both timed in the Graph500 search benchmark.

The reference implementation may be downloaded from www.graph500.org. The tarball is uncompressed and untarred as follows:

```
bzip2 -d graph500-2.1.4.tar.bz2
tar -xf graph500-2.1.4.tar
```

This will create a directory called graph500-2.1.4. In this directory there are several implementations, including MPI. To build the MPI version:

```
cd mpi
make
```

The *Makefile* for the reference implementation automatically assumes the use of the gnu compilers and that the MPI compiler wrapper *mpicc* is available in the user's path. The *Makefile* can be directly modified for alterations to these assumptions. No external libraries or dependencies are needed.

Five different executables result from the compile process, reflecting different ways of implementing the breadth-first search algorithm with graph500_mpi_simple being the standard level-

```
Usage: ./graph500_mpi_simple SCALE edgefactor
  SCALE = log_2(# vertices) [integer, required]
  edgefactor = (# edges) / (# vertices) = .5 * (average vertex degree) [integer, defaults to 16]
(Random number seed and Kronecker initiator are in main.c)
```

FIGURE 4.13

Usage reminder for the Graph500 search benchmark.

synchronized breadth-first search algorithm with a bitmap and two queues. This algorithm is explored in detail in Chapter 9. It requires at least one input to run and can take a second input. The usage for the benchmark shown in Fig. 4.13 appears in the event the user attempts to run the benchmark without arguments.

The first input supplies the code with the number of vertices:

$$N_{vertices} = 2^{scale} \tag{4.4}$$

The number of edges is given by product of the number of vertices and the edgefactor:

$$N_{edges} = edgefactor \times N_{vertices} \tag{4.5}$$

The default edgefactor is 16. Problem sizes in the Graph500 search benchmark are classified into six categories: toy, mini, small, medium, large, and huge. These are also referred to as levels 10–15, with level 10 being toy and level 15 huge. The scale factor for each of these and the associated memory requirements for the graph are given in Table 4.6.

An example execution of the Graph500_simple executable is as follows:

```
mpirun -np 16 ./graph500_mpi_simple 9
```

At this point the graph is constructed and the timing output for that graph is printed to screen, as shown in Fig. 4.14.

The timing for the breadth-first search kernel then prints to screen for each of the 64 roots, followed by a validation phase, as partially shown in Fig. 4.15.

Table 4.6 Problem Size Classes, Number of Vertices, and Memory Requirements for Graph500 Search Benchmark

Level	Scale	Size	Vertices (Billions)	Terabytes
10	26	Toy	0.1	0.02
11	29	Mini	0.5	0.14
12	32	Small	4.3	1.1
13	36	Medium	68.7	17.6
14	39	Large	549.8	141
15	42	Huge	4398.0	1126

```
graph_generation:          0.115093 s
construction_time:         0.224907 s
```

FIGURE 4.14

Graph generation statistics output from Graph500 search benchmark.

```
Running BFS 0
Time for BFS 0 is 0.007095
Validating BFS 0
Validate time for BFS 0 is 1.805835
TEPS for BFS 0 is 1.15464e+06
Running BFS 1
Time for BFS 1 is 0.000358
Validating BFS 1
Validate time for BFS 1 is 2.007691
TEPS for BFS 1 is 2.28912e+07
Running BFS 2
Time for BFS 2 is 0.000500
Validating BFS 2
Validate time for BFS 2 is 1.967331
TEPS for BFS 2 is 1.63852e+07
```

FIGURE 4.15

Partial output of the breadth-first search output for each of the 64 roots.

At the end of the graph traversal for each of the 64 roots, the final statistics for the graph500 search benchmark will print to screen, as shown in Fig. 4.16.

The Graph500 search benchmark does not output flops, but rather TEPS. This makes a comparison like that between HPL and HPCG difficult for the Graph500 search benchmark. However, two important trends for this benchmark are noticeable. First, while the HPL benchmark continues to show exponential improvement on newer supercomputers, the Graph500 search benchmark performance has gone flat. This is illustrated in Fig. 4.17, where the best performance of the Graph500 search benchmark is plotted as a function of time. This can be compared to Fig. 1.2, where the HPL performance for the Top 500 list continues to grow exponentially while the Graph500 performance has flat-lined.

The second noticeable trend is that the effective giga-traversed edges per second (GTEPS) per core is much lower for distributed-memory architectures than for shared memory. This is illustrated in Fig. 4.18, where the effective GTEPS/core in the best distributed and shared-memory results for each of problem scales 31−34 are plotted and the problem scale = $\log_2(N_{vertices})$.

4.9 MINIAPPLICATIONS AS BENCHMARKS

While benchmarks continue to serve an important role in the HPC community, there are many criticisms about their validity in fully capturing real application behavior. One of the principal concerns is that HPC benchmarks are too simple to assess a supercomputer's performance properly with respect to a dynamic application. HPC benchmarks generally aim to be specific to a small subset of independent HPC system performance attributes by design. To complement HPC benchmarking efforts and better capture real application behavior, many in the HPC community have turned to using miniapplications.

```
SCALE:                          9
edgefactor:                     16
NBFS:                           64
graph_generation:               0.115093
num_mpi_processes:              16
construction_time:              0.224907
min_time:                       0.000169039
firstquartile_time:             0.000205517
median_time:                    0.000227094
thirdquartile_time:             0.000342607
max_time:                       0.0959749
mean_time:                      0.00589841
stddev_time:                    0.019746
min_nedge:                      8192
firstquartile_nedge:            8192
median_nedge:                   8192
thirdquartile_nedge:            8192
max_nedge:                      8192
mean_nedge:                     8192
stddev_nedge:                   0
min_TEPS:                       85355.6
firstquartile_TEPS:             2.39107e+07
median_TEPS:                    3.60732e+07
thirdquartile_TEPS:             3.98605e+07
max_TEPS:                       4.84623e+07
harmonic_mean_TEPS:             1.38885e+06
harmonic_stddev_TEPS:           585773
min_validate:                   1.72823
firstquartile_validate:         1.86437
median_validate:                1.91839
thirdquartile_validate:         2.00359
max_validate:                   2.11599
mean_validate:                  1.92975
stddev_validate:                0.0889681
```

FIGURE 4.16

Final statistical output from the Graph500 search benchmark.

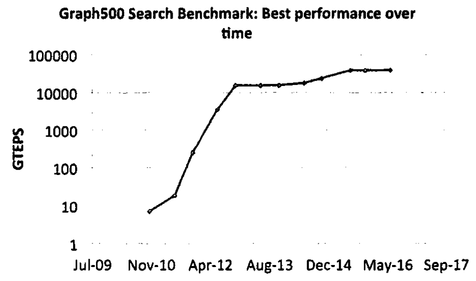


FIGURE 4.17

The Graph500 best performance as a function of time. Performance has gone flat, while the HPL performance continues to grow exponentially.

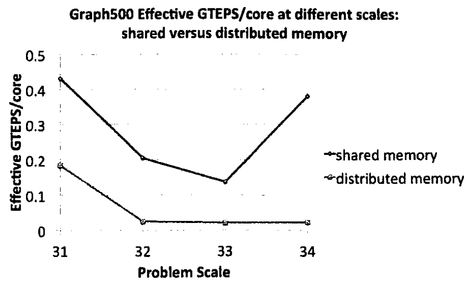


FIGURE 4.18

Results from the June 2016 Graph500 list comparing the effective GTEPS/core for the best results in shared memory and distributed memory at problem scales 31–34.

As the name implies, miniapplications are smaller versions of real applications. They originate from a large number of scientific disciplines and are generally much longer than HPC benchmarks. They do not generally output any standardized metric like flops, GUPS, TEPS, or degrees of freedom per second (DOFS), but do provide time to solution for various kernels as well as strong and weak scaling information. Table 4.7 provides an overview of some common miniapplications from the Mantevo suite [12] organized by Michael Heroux (HPCG benchmark cocreator) and Richard Barrett.

Miniapplications fulfill several roles that are difficult for standard HPC benchmarks. They enable large application developers to interact with a broader software engineering community by producing simplified, smaller, open-source versions of their application for outside scrutiny and optimization. Miniapplications also serve an important role in testing emerging programming models outside the scope of conventional parallel programming APIs like MPI and OpenMP. Miniapplications are well suited for performing scaling studies, especially in the context of dynamic simulations and on emerging hardware architectures. Finally, miniapplications are sufficiently complex yet small enough to explore the parameter and interaction space of memory, network, accelerators, and processor elements.

Table 4.7 Some Characteristics of Miniapplications from the Mantevo Suite

Mini-Application	Approximate Line Count	Parallelism			Language		
		MPI	OpenMP	Other	Fortran	C	C++
MiniAMR	9,400	X				X	
MiniFE	14,200	X	X	CUDA, Cilk			X
MiniGhost	12,770	X	X	OpenACC	X		
MiniMD	6,500	X	X	OpenCL, OpenACC			X
CloverLeaf	9,300	X	X	OpenACC, CUDA	X	X	
TeaLeaf	6,500	X	X	OpenCL	X		

The Mantevo suite contains a large number of open-source miniapplications from a wide array of application domains, including those listed below.

- MiniAMR—a miniapplication for exploring adaptive mesh refinement and dynamic execution with refinement and coarsening of meshes driven by objects passing through the mesh.
- MiniFE—a miniapplication for finite element codes.
- MiniGhost—a miniapplication for exploring halo exchange in the context of a finite differencing application on uniform 3D mesh.
- MiniMD—a miniapplication based on a molecular dynamics workload.
- Cloverleaf—a miniapplication for solving compressible Euler equations.
- TeaLeaf—a miniapplication based on a workload for solving linear heat conduction equation.

Some of these miniapplications are revisited in the context of the software libraries discussion in Chapter 10.

In addition to the Mantevo suite, a large number of miniapplications are maintained at the many supercomputing centers around the world. These miniapplications often complement standard HPC benchmarks by playing a significant role in procurement decisions. Consequently, they have significant supercomputing vendor involvement as well. As an example, in the collaboration of Oak Ridge, Argonne, and Livermore US National Laboratories (CORAL) to procure two 150 petaflops machines, results from over 25 miniapplications were requested from hardware vendors [13] in addition to several of the benchmarks mentioned in this chapter.

4.10 SUMMARY AND OUTCOMES OF CHAPTER 4

- Benchmarking is a way to measure the performance of a supercomputer empirically. A benchmark provides some standardized type of workload that may vary in size or input dataset.
- Computational benchmark workloads come in two types: synthetic, where workloads are designed and created to impose a load on a specific component in the system; and application, where the workload is derived from a real-world application.
- Good benchmarks are relevant and meaningful to the target application domain, applicable to a broad spectrum of hardware architectures, adopted by both users and vendors, and enable comparative evaluation.
- Early benchmarks include the floating-point-intensive Whetstone benchmark and the integer-oriented Dhrystone benchmark.
- The Linpack benchmark solves a dense, regular system of linear equations and provides an estimate of a system's effective floating-point performance.
- The HPL benchmark is used for ranking supercomputers in the Top 500 list.
- HPL is part of the HPC Challenge benchmark suite that contains seven widely used HPC benchmarks.
- The HPCG benchmark is meant to complement the HPL benchmark in exploring memory and data-access patterns in application workloads that are not well represented by HPL. The workload in HPCG centers on a sparse system of linear equations arising from the discretization of a 3D Laplacian partial differential equation with a 27-point stencil.

- HPCG performance continues to be, at best, a very small fraction of HPL performance on even the fastest supercomputers in the Top 500 list.
- NPB is a series of small self-contained programs that encapsulate the performance attributes of a large computational fluid dynamics application.
- NPB started as a pencil-and-paper benchmark, but later reference implementations became the benchmark itself in NPB iterations.
- The Graph500 benchmark is intended to represent data-intensive workloads.
- The Graph500 search benchmark implements the breadth-first search algorithm and reports TEPS as a key metric.
- Graph500 benchmark performance has gone flat even while HPL benchmark performance continues to grow exponentially.
- To complement HPC benchmarking efforts and better capture real application behavior, many in the HPC community have turned to using miniapplications.
- Miniapplications fulfill several roles that are difficult for standard HPC benchmarks, including exploring the parameter and interaction space of memory, network, accelerators, and processor elements, especially in terms of emerging hardware and programming models.

4.11 EXERCISES

1. Run the HPL benchmark on an accessible supercomputer and an available laptop. Tune the input parameters independently for each system to get the best possible performance. For what matrix size does the supercomputer give the best HPL performance? At what matrix size does the laptop give the best HPL performance? Explain your results in terms of the system architecture and memory characteristics of HPL.
2. Run the HPCG benchmark on an accessible supercomputer. Compare the peak HPCG performance versus the peak HPL performance. Which performs best and why?
3. Compile and run the HPC Challenge benchmark suite on an accessible supercomputer and an available laptop. Provide a table with the final results (number and units) of each of the seven problems. Your table should have two columns: test name, and a numeric value of a certain metric with its units. Pick only one metric for each problem. Compare the performance between the supercomputer and the laptop.
4. Run the Graph500 benchmark on an accessible supercomputer. Plot the performance of the Graph500 in GTEPS as a function of graph size. What is the biggest graph problem that you can run on the supercomputer?
5. Explore the performance of the discrete 3D Fourier transform on an accessible supercomputer using the FT NPB. Plot the performance in gigaflops as a function of problem size. What is the peak gigaflops achieved for FT compared with the peak gigaflops achieved for the HPL benchmark on the same supercomputer?

REFERENCES

[1] Wikipedia, ENIAC, [Online]. https://en.wikipedia.org/wiki/ENIAC.
[2] R. Longbottom, History of Whetstone, [Online]. http://www.roylongbottom.org.uk/whetstone.htm.

[3] Standard Performance Evaluation Corporation, SPEC CPU, 2006 [Online], https://www.spec.org/cpu2006/.
[4] Netlib, Linpack FAQ, [Online]. http://www.netlib.org/utk/people/JackDongarra/faq-linpack.htm.
[5] LAPACK, [Online]. http://www.netlib.org/lapack/.
[6] Innovative Computing Laboratory, The University of Tennessee, HPC Challenge Benchmark Suite, [Online]. http://icl.cs.utk.edu/hpcc/.
[7] NASA, NAS Parallel Benchmarks, [Online]. http://www.nas.nasa.gov/publications/npb.html.
[8] Top500, Top500 List, [Online]. https://www.top500.org/lists/.
[9] HPCG, HPCG Benchmark, [Online]. http://www.hpcg-benchmark.org/.
[10] Computational Research, Berkeley Laboratory, HPGMG Performane Results, [Online]. https://crd.lbl.gov/departments/computer-science/PAR/research/hpgmg/results/.
[11] Graph500, Graph500, [Online]. http://graph500.org/.
[12] M. Heroux, Mantevo Suite of Mini Apps, [Online]. https://mantevo.org.
[13] Lawrence Livermore National Laboratory, Coral Benchmarks, [Online]. https://asc.llnl.gov/CORAL-benchmarks.

CHAPTER

THE ESSENTIAL RESOURCE MANAGEMENT 5

CHAPTER OUTLINE

5.1 Managing Resources **142**
5.2 The Essential SLURM **146**
5.2.1 Architecture Overview 147
5.2.2 Workload Organization 148
5.2.3 SLURM Scheduling 149
5.2.3.1 Gang Scheduling 149
5.2.3.2 Preemption 149
5.2.3.3 Generic Resources 150
5.2.3.4 Trackable Resources 150
5.2.3.5 Elastic Computing 150
5.2.3.6 High-Throughput Computing 150
5.2.4 Summary of Commands 151
5.2.4.1 srun 151
5.2.4.2 salloc 160
5.2.4.3 sbatch 161
5.2.4.4 squeue 162
5.2.4.5 scancel 163
5.2.4.6 sacct 164
5.2.4.7 sinfo 165
5.2.5 SLURM Job Scripting 166
5.2.5.1 Script Components 166
5.2.5.2 MPI Scripts 167
5.2.5.3 OpenMP Scripts 167
5.2.5.4 Concurrent Applications 167
5.2.5.5 Environment Variables 169
5.2.6 SLURM Cheat Sheet 171
5.3 The Essential Portable Batch System **172**
5.3.1 Portable Batch System Overview 172
5.3.2 Portable Batch System Architecture 173
5.3.3 Summary of PBS Commands 174
5.3.3.1 qsub 174
5.3.3.2 qdel 180

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00005-8

5.3.3.3 qstat 180
5.3.3.4 tracejob 182
5.3.3.5 pbsnodes 183
5.3.4 PBS Job Scripting 184
5.3.4.1 OpenMP Jobs 184
5.3.4.2 MPI Jobs 185
5.3.4.3 Environment Variables of Interest 185
5.3.5 PBS Cheat Sheet 186
5.4 Summary and Outcomes of Chapter 5 .... **187**
5.5 Questions and Problems **189**
References **190**

5.1 MANAGING RESOURCES

Supercomputer installation frequently represents a significant financial investment by the hosting institution. However, the expenses do not stop after the hardware acquisition and deployment is complete. The hosting data center needs to employ dedicated system administrators, pay for support contracts and/or a maintenance crew, and cover the cost of electricity used to power and cool the machine. Together these are referred to as "cost of ownership". The electricity cost is frequently overwhelming for large installations. A commonly quoted average is over US$1 million for each megawatt of power consumed per year in the United States; in many other countries this figure is much higher. It is not surprising that institutions pay close attention to how supercomputing resources are used and how to maximize their utilization.

Addressing these concerns, resource management software plays a critical role in how supercomputing system resources are allocated to user applications. It not only helps to accommodate different workload sizes and durations, but also provides uniform interfaces across different machine types and their configurations, simplifying access to them and easing (at least some) portability concerns. Resource management middleware provides mechanisms by which computing systems may be made available to various categories of users (including those external to the hosting institution, for example via collaborative environments such as the National Science Foundation XSEDE [1]) with accurate accounting and charging for the resource use. Resource management tools are an inherent part of the high performance computing (HPC) software stack. They perform three principal functions: resource allocation, workload scheduling, and support for distributed workload execution and monitoring. Resource allocation takes care of assigning physical hardware, which may span from a fraction of the machine to the entire system, to specific user tasks based on their requirements. Resource managers typically recognize the following resource types.

- *Compute nodes*. Increasing the number of nodes assigned to a parallel application is the simplest way to scale the size of the dataset (such as the number of grid points in a simulation domain) on which the work is to be performed, or reduce the execution time for a fixed workload size. Node count is therefore one of the most important parameters requested when scheduling an application launched on a parallel machine. Even single physical computers may include various node types; for example differing in memory capacity, central processing unit (CPU) types and clock frequency, local storage characteristics, available interconnects, etc. Properly configured resource managers permit selection of the right kind of node for the job, precluding assigning resources that will likely go unused.

- *Processing cores (processing units, processing elements).* Most modern supercomputer nodes feature one or more multicore processor sockets, providing local parallelism to applications that support it through multithreading or by accommodating several concurrent single-threaded processes. For that reason, resource managers provide the option of specifying *shared* or *exclusive* allocation of nodes to workloads. Shared nodes are useful in situations where already assigned workloads would leave some of the cores unoccupied. By coscheduling different processes on the remaining cores, better utilization may be achieved. However, this comes at a cost: all programs executing on the shared node will also share access to other physical components, such as memory, network interfaces, and input/output (I/O) buses. Users who perform careful benchmarking of their applications are frequently better off allocating the nodes in exclusive mode to minimize the intrusions and resulting degradation due to contention caused by unrelated programs. Exclusive allocation can also be used for programs that rely on the affinity of the executing code to specific cores to achieve good performance. For example, programs that rely on lowest communication latency may want to place the message sending and receiving threads on cores close to the PCI express bus connected to the related network card. This may not be possible when multiple applications enforce their own, possibly conflicting, affinity settings at the same time.
- *Interconnect.* While many systems are built with only one network type, some installations explicitly include multiple networks or have been expanded or modernized to take advantage of different interconnect technologies, such as GigE and InfiniBand architecture in combination. Selection of the right configuration depends on the application characteristics and needs. For example, is the program execution more sensitive to communication latency, or does it need as much communication bandwidth as possible? Can it take advantage of channel bonding using different network interfaces? Often the answer may be imposed by the available version of the communication library with which the application has been linked. For example, it is common to see message-passing interface (MPI) installations with separate libraries supporting InfiniBand and Ethernet if both such network types are available. Selecting a wrong network type will likely result in less efficient execution.
- *Permanent storage and I/O options.* Many clusters rely on shared file systems that are exported to every node in the system. This is convenient, since storing a program compiled on the head node in such a file system will make it available to the compute nodes as well. Computations may also easily share a common dataset, with modifications visible to the relevant applications already during their runtime. However, not all installations provide efficient high-bandwidth file systems that are scalable to all machine resources and can accommodate concurrent access by multiple users. For programs performing a substantial amount of file I/O, localized storage such as local disks of individual nodes or burst buffers (fast solid-state device pools servicing I/O requests for predefined node groups) may be a better solution. Such local storage pools are typically mounted under a predefined directory path. The drawback is that the datasets generated this way will have to be explicitly moved to the front-end storage after job completion to permit general access (analysis, visualization, etc.). Since there is no single solution available, users should consult

local machine guides to determine the best option for their application and how it can be conveyed to the resource management software.

- *Accelerators.* Heterogeneous architectures that employ accelerators (graphics processing units (GPUs), many integrated cores (MICs), field programmable gate array modules, etc.) in addition to main CPUs are a common way to increase the aggregate computational performance while minimizing power consumption. However, this complicates resource management, since the same machine may consist of some nodes that are populated with accelerators of one type, some nodes that are populated with accelerators of a different type, and some nodes that do not contain any accelerating hardware at all. Modern resource managers permit users to specify parameters of their jobs so that the appropriate node types are selected for the application. At the same time, codes that do not need accelerators may be confined to regular nodes as much as possible for best resource utilization over multiple jobs.

Resource managers allocate the available computing resources to *jobs* specified by users. A job is a self-contained work unit with associated input data that during its execution produces some output result data. The output may be as little as a line of text displayed on the console, or a multiterabyte dataset stored in multiple files, or a stream of information transmitted over the local or wide area network to another machine. Jobs may be executed *interactively,* involving user presence at the console to provide additional input at runtime as required, or use *batch processing* where all necessary parameters and inputs for job execution are specified before it is launched. Batch processing provides much greater flexibility to the resource manager, since it can decide to launch the job when it is optimal from the standpoint of HPC system utilization and is not hindered by the availability of a human operator, for example at night. For this reason, interactive jobs on many machines may be permitted to use only a limited set of resources.

Jobs may be monolithic or subdivided into a number of smaller *steps* or *tasks.* Typically each task is associated with the launch of a specific application program. In general, individual steps do not have to be identical in terms of used resources or duration of execution. Jobs may also mix parallel application invocations with instantiations of single-threaded processes, dramatically changing the required resource footprint. An example is a job that first preprocesses input data, copying them to storage local to its execution nodes, then launches the application that gives high-bandwidth access to the data, and finally copies the output files to shared storage using shell commands.

Pending computing jobs are stored in job *queues.* The job queue defines the order in which jobs are selected by the resource manager for execution. As the computer science definition of the word suggests, in most cases it is "first in, first out" or "FIFO", although good job schedulers will relax this scheme to boost machine utilization, improve response time, or otherwise optimize some aspect of the system as indicated by the operator (user or system administrator). Most systems typically use multiple job queues, each with a specific purpose and set of scheduling constraints. Thus one may find an interactive queue solely for interactive jobs. Similarly, a debug queue may be employed that permits jobs to run in a restricted parallel environment that is big enough to expose problems when running on multiple nodes using the same configuration as the production queue, yet small enough that the pool of nodes for production jobs may remain substantially larger. Frequently there are multiple production queues available, each with a different maximum execution time imposed on jobs or total job size (short versus long, large versus small, etc.). With hundreds to thousands of jobs with different properties pending in all queues of a typical large system, it is easy to see why scheduling algorithms are critical to achieving high job throughput. Common parameters that affect job scheduling include the following.

- *Availability of execution and auxiliary resources* is the primary factor that determines when a job can be launched.
- *Priority* permits more privileged jobs to execute sooner or even preempt currently running jobs of lower priority.
- *Resources allocated to the user* determines the long-term resource pool a specific user may consume while his or her account on the machine remains active.
- *Maximum number of jobs* that a user is permitted to execute simultaneously.
- *Requested execution time* estimated by the user for the job.
- *Elapsed execution time* may cause forced job termination or impact staging of pending jobs for upcoming execution.
- *Job dependencies* determine the launching order of multiple related jobs, especially in producer—consumer scenarios.
- *Event occurrence*, when the job start is postponed until a specific predefined event occurs.
- *Operator availability* impacts the launch of interactive applications.
- *Software license availability* if a job is requesting the launch of proprietary code.

Resource managers are equipped with optimized mechanisms that enable efficient launching of thousands or more processes across a comparable number of nodes. Naïve approaches, such as repeated invocation of a remote shell, will not yield acceptable results at scale due to high contention when transferring multiple programs' executables to the target nodes. Job launchers employ hierarchical mechanisms to alleviate the bandwidth requirements and exploit network topology to minimize the amount of data transferred and overall launch time. Resource managers must be able to terminate any job that exceeds its execution time or other resource limits, irrespective of its current processing status. Again, distributed termination should be efficient to release the allocated nodes to the pool of available nodes as quickly as possible. Finally, resource managers are responsible for monitoring application execution and keeping track of related resource usage. The actual resource utilization data is recorded to enable accounting and accurate charging of users for their cumulative system resource usage.

A number of resource management suites have been created that differ in their features, capabilities, and adoption level. The software commonly used today includes the following.

- *Simple Linux Utility for Resource Management (SLURM)* [2] is a widely used free open-source package.
- *Portable Batch System (PBS)* [3] was originally available as proprietary code as well as several open implementations with compatible application programming interface and commands.
- *OpenLava* [4] is an open-source scheduler based on the Platform Load Sharing Facility and originally developed at the University of Toronto.
- *Moab Cluster Suite* [5], based on the open-source Maui Cluster Scheduler, is a highly scalable proprietary resource manager developed by Adaptive Computing Inc.
- *LoadLeveler* [6], currently known as the Tivoli Workload Scheduler LoadLeveler, is a proprietary IBM product originally targeting systems running the AIX operating system (OS) but later ported to POWER and x86-based Linux platforms.

- *Univa Grid Engine* [7] uses technology originally developed by Sun Microsystems and Oracle that supports multiple platforms and OSs.
- *HTCondor* [8], formerly known just as Condor, is an open-source framework for coarse-grain high-throughput computing.
- *OAR* [9] provides database-centered resource and task management for HPC clusters and some classes of distributed systems.
- *Hadoop Yet Another Resource Negotiator (YARN)* [10] is a broadly deployed scheduler specifically tailored to MapReduce applications, discussed in detail in Chapter 19.

Unfortunately there is no common standard specifying the command format, language, and configuration of resource management. Every system mentioned above uses its own interface and supports different sets of capabilities, although the basic functionality is essentially similar. Thus two widely used examples of resource managers are described here in detail, SLURM and PBS. Both have particularly broad adoption in the HPC community. These sections are presented in tutorial form to build the reader's skill-set.

5.2 THE ESSENTIAL SLURM

SLURM is an open-source, modular, extensible, scalable resource manager and workload scheduling software for clusters and supercomputers running Linux or other Unix-compatible OS. Its origins date back to 2001, when a small team of developers started by Morris Jette at Lawrence Livermore National Laboratory originated work on advanced scheduling systems for HPC. Since that time SLURM development has grown significantly, extending to nearly 200 contributors as well as multiple institutions, including SchedMD LLC (currently the core company responsible for its development, support, training, and consulting services), Linux NetworX, Hewlett–Packard, Groupe Bull, Cray, Barcelona Supercomputing Center, Oak Ridge National Laboratory, Los Alamos National Laboratory, Intel, Nvidia, and many others. In June 2014 SLURM was among the most dominant resource management systems, being utilized in approximately 60% of machines in the Top 500 list [11].

The popularity of SLURM is in no small part due to its impressive list of operational features. As an open-source solution it is available and affordable to even the smallest computing centers and schools. Its core functionality may be extended using plugins written in C or Lua, thus providing complex configuration options and support for various interconnect types, scheduling algorithms, MPI implementations, accounting, and more. SLURM scales to the largest systems in use today, including the fastest supercomputer of 2016, Sunway TaihuLight, with its 40,000 CPUs (over 10 million cores). Five of the top 10 machines are managed by SLURM. It can handle up to a 1000 job submissions and 500 job executions per second. A number of strategies are available to optimize power consumption, ranging from the ability to specify clock frequency for CPUs to powering down unused nodes completely—an important feature when power draw for the largest platforms may exceed 10 MW. Adjusted power levels can be entered in job records to account more accurately for resource usage. Single points of failure are eliminated through the use of multiple backup daemons, permitting the affected applications to continue running and request resources to replace those that fail. Network topology factors into resource allocation to minimize communication latency when it is critical to application execution. SLURM maintains detailed architectural information about each component node, including distribution of cores across nonuniform memory access domains and hyperthread affinities. The user may utilize these parameters to optimize binding of tasks to resources. Job sizes are not necessarily fixed over their execution time; they may grow or shrink in accordance with the specified size and time limits. Sophisticated scheduling algorithms are available, including gang scheduling and preemption. Control over scheduling policies is enabled through constraints specified by the user, a bank account, or quality of service metrics. Finally, SLURM integrates support for execution on heterogeneous components, such as GPUs, MIC processors, and other accelerators. An optional database may be used to store each job's execution profile, detailing CPU, memory, network, and I/O usage, providing the means for postmortem analysis and optimization of system allocation in the future.

5.2.1 ARCHITECTURE OVERVIEW

To support its extensive functionality, SLURM employs a collection of daemons (programs continuously running in the background) to interpret user commands and distribute work to individual nodes in the system. Similar arrangements are commonly used by other cluster resource management systems. Users, including programmers and system administrators, issue commands on one of the head nodes. These commands typically communicate with local control daemons *slurmctld*, which relay specific management tasks to the *slurmd* daemons running on the compute nodes. Some commands may directly interact with *slurmd* backends. Each *slurmd* daemon listens to a network connection to accept an incoming work item, execute it, return completion status, and wait for another work unit. These daemons are organized hierarchically to optimize communication and provide fault tolerance, as illustrated in Fig. 5.1. SLURM may optionally support a performance collection database, shown in the figure as an external storage component marked *db* and managed by a dedicated daemon, *slurmdbd*. *Slurmdbd* may also connect to other machines to provide a central recording of accounting information for multiple clusters that run the SLURM software suite.

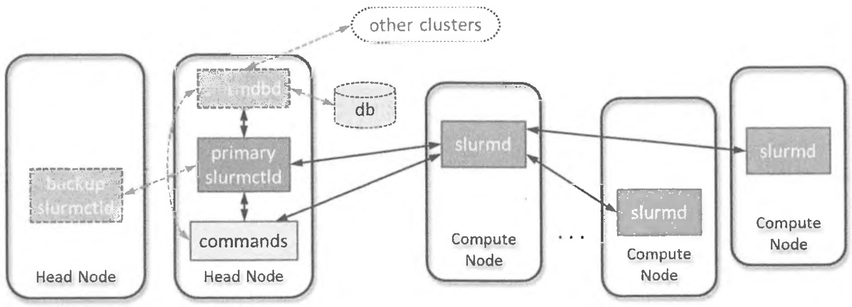


FIGURE 5.1

Simplified architecture of SLURM. Components framed by *dashed lines* are optional.

5.2.2 WORKLOAD ORGANIZATION

One of the primary resource types managed by SLURM is the compute node. Nodes are divided into logical sets called *partitions*. Partitions in SLURM represent individual job queues and thus impose specific constraints on user jobs. Depending on the prevalent characteristics of computational workloads and user needs, the cluster administrator may decide to create completely disjoint or overlapping partitions. The latter may be useful to permit the allocation of all available execution resources to certain, usually severely constrained, job types.

The scheduler assigns the available nodes in the partition to the highest-priority eligible jobs until the pool of available nodes is exhausted. The individual tasks composing a job, called *job steps*, may utilize the entire set of nodes allocated to the parent job or only a fraction. The example in Fig. 5.2 shows a 20-node cluster that has been partitioned into two disjoint node sets, `Partition 1` and `Partition 2`. As illustrated in the figure, `Job 1` has been assigned all nodes in `Partition 1` and all are currently utilized by `Job Step 1`. In `Partition 2` the scheduler designated only 9 out of 12 available nodes for `Job 2` and 8 of them are in use by two concurrent job steps, `Job Steps 5` and `6` (they could be a physics simulation application and connected visualization engine executing in parallel). The remaining three nodes in `Partition 2` could be allocated to another job concurrent with `Job 2` as long as its resource constraints can be satisfied. A typical system would use more meaningful partition names indicative of their function in the system, such as `debug` or `main`. Similarly, good practice calls on users to label their jobs in a way that permits easy identification of their purpose and configuration variant.

SLURM uses the concept of *job arrays* to provide a highly efficient means for submission and management of collections of similar jobs. While their initial parameters, such as time limit or size, have to be identical, they may be changed later on a per-array or per-job basis. Job arrays may only be batch processed.

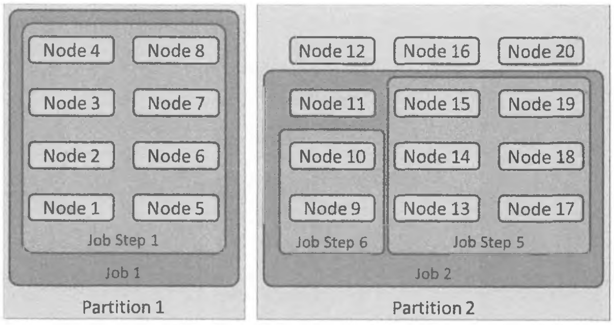


FIGURE 5.2

Relationships between partitions, jobs, and job steps in SLURM.

5.2.3 SLURM SCHEDULING

SLURM employs relatively simple default scheduling algorithms to comply with its design goals of efficiency and simplicity. Whenever a job is submitted or completed, or system configuration changes, only a limited and predefined number of jobs at the front of the queue will be considered for scheduling. This is called event-triggered scheduling. This algorithm is complemented by another that attempts to take into consideration all queued jobs before making scheduling decisions. Due to significantly increased overheads, the latter runs at much less frequent intervals. The scheduling algorithm marks the subset of highest-priority eligible jobs that in aggregate satisfy the available resource footprint as pending for execution. As long as there are any pending jobs in a partition, scheduling for that partition is disabled.

SLURM also provides a *backfill* scheduler plugin that can considerably improve the overall arrangement of job execution compared to the priority-based FIFO scheduling. For example, low-priority jobs requesting a significant amount of resources could be delayed indefinitely in the queue if the influx of small, higher-priority jobs is large. Due to a larger number of jobs considered for execution, the system utilization may also improve. The backfill scheduler will attempt to start lower-priority jobs if this will not delay the expected start time of any of the higher-priority jobs. Making accurate scheduling decisions relies heavily on job completion estimates, defined as the queue configuration parameters and submitted as wallclock limit estimates with individual jobs. Thus the administrators of many systems recommend that their users specify these constraints as accurately as possible.

Job scheduling in SLURM is a complex topic, with many additional improvements and scheduler variants available through plugins. The following is a brief discussion of some of the more prominent concepts.

5.2.3.1 Gang Scheduling

Gang scheduling supports a scheduling approach in which two or more jobs with similar characteristics are allocated the same set of resources. These jobs are then executed in an alternating fashion so that only one of them obtains the exclusive access to the resources at a time. The time for which a single job retains access to the resources, or a *timeslice*, is a configurable parameter. This scheduling mode permits shorter jobs to be started ahead of longer jobs as long as there are available resources, instead of being forced to wait in the queue behind the longer job. As a result they may be started (and finished) earlier, increasing the overall throughput of the system.

SLURM spawns a dedicated timeslicer thread that prevents starvation of gang-scheduled jobs. The timeslicer wakes up periodically (at the start of each timeslice period) and checks for suspended jobs. If there are any, the currently running jobs are moved to the end of queue. The timeslicer then calculates a new allocation for the partition by scanning the queue for suspended jobs which have been waiting longest to run. If there are other active jobs that can be run concurrently with the newly computed allocation, they are added to it. All other currently running jobs that do not fit into the new allocation are suspended.

5.2.3.2 Preemption

Closely related to gang scheduling is preemption, or stopping lower-priority jobs to permit the execution of higher-priority jobs. Preemption is implemented using a variant of gang scheduling.

Whenever a high-priority job receives a resource allocation that overlaps the allocation already assigned to one or more low-priority jobs, the affected low-priority jobs are preempted. They may resume once the high-priority job completes or, in newer versions of SLURM, be requeued and started using a different set of resources.

5.2.3.3 Generic Resources

Generic resources (GRES) in SLURM terminology refer to other hardware devices associated with nodes, most commonly accelerators. SLURM currently supports Nvidia GPUs and Intel MICs through a plugin mechanism. As there is no default configuration available, the system administrator has to specify the resource name, count, CPUs that may access the resource, device type, and file system pathname that can be used to access or exclusively allocate the device. Only the permitted CPUs may access the device, even though there may be no physical counterindications to disable access for the remaining CPUs in the node. Unlike other execution resources, GRES allocated to a job will not become available to other jobs when that job gets suspended. The job steps may request fewer GRES than the amount allocated to the parent job (by default they are allocated all the GRES that the job holds). This permits easy partitioning of GRES among concurrent jobs steps.

5.2.3.4 Trackable Resources

SLURM provides additional options to track the use of or enforce custom constraints on various kinds of resources. Such trackable resources (TRES) are identified by their types and names; examples include burst buffers, CPUs, energy, GRES, licenses, memory, and nodes. This feature helps establish more accurate formulas to bill for computer usage in which predefined weights may be assigned to each TRES type.

5.2.3.5 Elastic Computing

Elastic computing refers to a scenario in which the overall resource footprint available in a system or consumed by a specific job can grow or shrink on demand. This usually relies on external cloud computing services, where the local cluster provides only part of the resource pool available to all jobs. However, elastic computing may also be implemented on standalone clusters.

Elastic computing may improve power efficiency by explicitly turning off nodes that are not in use. These nodes will be restored to normal operation as soon as there are any jobs assigned to them. To prevent power surges, which are inevitable when powering up or down large groups of nodes, SLURM gradually changes power consumption at a configurable rate. This typically requires CPU throttling support built into the OS kernel on the affected nodes. The power-saving algorithms drive the node provisioning logic in coordinating reservation and relinquishing the external nodes to the cloud as needed.

5.2.3.6 High-Throughput Computing

SLURM provides rudimentary support for high-throughput computing, in which large numbers of relatively small, loosely coupled jobs are launched over an extended period of time. A correctly tuned SLURM system may execute as many as 500 simple batch jobs per second (sustained), with bursts significantly exceeding this number. SLURM high-throughput job selection logic has been significantly optimized, retaining roughly half of the original scheduling code.

5.2.4 SUMMARY OF COMMANDS

The purpose of this subsection is to familiarize the reader interested in using a system equipped with the SLURM resource manager with the basic commands to perform job submission, job status retrieval, system status query, and basic management of jobs. Commands primarily targeting system administration are beyond the scope of this presentation. Along with each command description listed are the most frequently used options (both short and long forms are provided if available) and usage examples. The option syntax used below shows literal parameter names and operators except for the following cases:

- angle brackets, "<" and ">", signify a parameter name, which may expand to a number or a string depending on the context
- square brackets, "[" and "]", denote an optional entry
- braces, "{" and "}", encompass a list to describe selection of one of the items in that list.

SLURM commands start with lower-case "s" and include the following.

5.2.4.1 srun

srun [<options>] <executable> [<arguments>]

The srun command is used to start parallel jobs or job steps on a cluster. If the resources to run the job have not been allocated yet (for example, the command is executed on the head node's terminal), the resource allocation will be performed first. If invoked from an already started job, such as the job's batch script, srun starts a new job step. If the resources to start the job are available, the job is started immediately, otherwise the command blocks until the resources become available.

The list of options presented below is comprehensive, but in no way exhaustive. Many of these options also apply to resource allocation used by other SLURM commands.

-N or **--nodes=<min_nodes>[-<max_nodes>]**

This allocates nodes for a job to be executed. The number of nodes has to be at least **min_nodes** but not exceed **max_nodes**. The numbers may be followed by the suffix "k" or "m" to signify a multiplier of 1024 (2^{10}) or 1,048,576 (2^{20}). SLURM will allocate as many nodes in the range specified as possible without causing additional delays.

Example

```
srun -N1 /bin/bash
```

This starts an interactive shell on one of the compute nodes for the default period of time.

-n or **--ntasks=<number_of_tasks>**

-c or **--cpus-per-task=<number_of_cpus>**

The **ntasks** option specifies the number of tasks (processes) to run and requests allocation of a sufficient number of nodes for them. By default one task per node is started, unless overridden by the

cpus-per-task option, which defines the maximum number of cores assigned to each process. The latter may be used to launch multithreaded processes.

Example

```
srun -n4 -c8 my_app
```

This launches four processes using executable my_app, each limited to eight threads of execution. If the cluster has 16-core nodes, 2 nodes will be allocated for the job, unless the **exclusive** option is used (see below).

--mincpus=<number_of_cpus>

This allocates nodes for job that have at least number_of_cpus cores per node available.

Example

```
srun -n4 -c8 --mincpus=32 my_app
```

This will place all four instances of my_app on a single node.

-B or **--extra-node-info=<sockets_per_node>[:<cores_per_socket>[:<threads_per_core>]]**

--cores-per-socket=<number_of_cores>

--sockets-per-node=<number_of_sockets>

--threads-per-core=<number_of_threads>

The first form allocates nodes with a specific number of sockets (physical processors), and optionally given a count of cores per socket and threads per core. The last parameter applies to architectures that permit concurrent threads effectively to share execution units, such as Intel processors with hyperthreading. The remaining three options enable independent specification of each parameter.

Example

```
srun -N1 -B2:4 my_app
srun -N1 --cores-per-socket=4 --sockets-per-node=2 my_app
```

Both examples are equivalent, and will allocate one node for application my_app with at least two physical CPUs each containing at least four cores.

-m or **--distribution=<node_distr>[:<socket_distr>[:<core_distr>]][,{Pack,NoPack}]**

This specifies different distribution modes of the job's tasks across system resources. It may have dramatic implications for application performance, e.g., due to grouping related threads on topologically close resources and separating unrelated tasks. The option argument contains up to three entries separated by colons, ":", that determine process assignment to nodes, sockets, and cores, respectively. Only the first entry (node distribution) is required. The argument may optionally contain the **Pack** or **NoPack** directive, which either directs the allocator to pack the tasks on the nodes as tightly as possible or forces as even a task distribution as possible. The node distribution parameters are as follows:

- * to accept the default distribution, typically **block**
- **block** will try to assign consecutive tasks to the same node before moving to the next node
- **cyclic** distributes consecutive tasks over consecutive nodes in a round-robin fashion
- **plane=<size>** distributes the processes in blocks of specified **size**; after placing block of **size** processes on one node, it moves to the next node to assign the next block, and so on
- **arbitrary** mode distributes the tasks in the order specified in the environment variable SLURM_HOSTFILE, defaulting to **block** if the variable is unspecified.

  The supported socket and core distribution parameters are identical, and include:

- * default mode, which is **cyclic** for sockets and derived from the socket distribution for cores
- **block** assigns consecutive tasks to the same socket/core before moving to the next socket or core
- **cyclic** will assign CPUs consecutively from the same socket/core to the same task and from the next socket/core for the next task in a round-robin fashion.
- **fcyclic** or "full cyclic" assigns CPUs to tasks across consecutive sockets/cores in a round-robin fashion without trying to group them based on task boundary.

  Example

```
srun -n6 -c2 -m'block:cyclic' my_app
srun -n6 -c2 -mplane=2:fcyclic,NoPack my_app
```

If the first example is submitted on a machine equipped with dual quad-core processors (each core supporting single thread of execution), two nodes will be allocated for the job. Assuming the first socket of node 0 includes cores numbered 0–3 and the second cores 4–7, task 0 will run on cores 0 and 1, task 1 on cores 4 and 5, task 2 on cores 2 and 3, and task 3 on cores 6 and 7. The remaining tasks will be instantiated on node 1, with task 4 using cores 0 and 1 and task 5 cores 4 and 5.

Launching the second example on the same platform results in allocation of three nodes. Tasks 0 and 1 are assigned node 0, tasks 2 and 3 node 1, and tasks 4 and 5 node 3. Individual tasks within the node use cores 0 and 4 (first task) and 1 and 5 (second task).

-w or **--nodelist=<list_of_nodes>**

This requests specific nodes for job execution. The list may contain individual node names separated by commas, or node ranges. If **list_of_nodes** contains a "/" (forward slash character), it will be assumed to represent a path to file containing the node list. Note that if the specified node list is not sufficient to support the job, the system will attempt to allocate additional nodes as required.

Example

```
srun -wnode0[4-6],node08 -N6 my_app
```

This will allocate nodes 4, 5, 6, and 8 plus two more not explicitly specified nodes for the total of six required tasks.

--mem=<megabytes>

--mem-per-cpu=<megabytes>

This controls the allocation of physical memory. The first form specifies the total memory per node required for job execution. The value of zero specified in the job step invocation restricts that job step to memory allocated to the parent job. The second option is used to limit the amount of memory allocated to individual processors.

Only one of these options may be specified at a time.

Example

```
srun -N2 -c8 --mem-per-cpu=4096 my_app
```

Here "my_app" will be allocated 32 GB of memory (or 4 GB per core) on each of the two assigned nodes.

--hint=<type>

This allocates resources based on a literal hint describing the job's properties:

- **compute_bound** causes allocation of all cores in each socket with one thread per core
- **memory_bound** uses one core in each socket and one thread per core
- **[no]multithread** instructs the system (not) to use multiple threads per core, which could improve the performance of communication-intensive applications.

Example

```
srun -N48 --hint=compute_bound bh_mol
```

This will start the compute-bound application "bh_mol" on all cores of 48 assigned nodes.

--ntasks-per-core=<number>

--ntasks-per-socket=<number>

--ntasks-per-node=<number>

These set the upper bound for the number of tasks per core, socket, and node, respectively. The last option is useful for starting mixed MPI/OpenMP jobs which require that only one MPI process is created per node that utilizes multiple threads for increased local parallelism.

Example

```
srun -N16 --ntasks-per-node=16 mpirun my_sim
```

This will launch an MPI application on 16 nodes utilizing a total of 256 threads.

--multi-prog

This runs a job consisting of different programs with different arguments. A configuration file listing the applications with related arguments for each task is required. A path to that file replaces the usual executable name at the end of the **srun** command line. The syntax of this file is explained in the Section 5.2.5 discussing job scripting in detail.

--exclusive[=user]

-s or **--oversubscribe**

These affect resource undersubscription and oversubscription. The first option suppresses node sharing with other jobs. If the optional parameter **user** is specified, the node will not be shared with jobs submitted by other users, but may be available to jobs owned by the same user. When used for job step launch, each of the concurrently executing job steps is assigned a separate processor. If such assignment is not possible at the time of invocation, launch of the job step may be deferred.

The **oversubscribe** option permits the resource oversubscription with other jobs that may apply to nodes, sockets, cores, and hyperthreads depending on system configuration. Jobs enabling oversubscription may obtain their resource allocation sooner and thus be started earlier than in exclusive mode.

Example

```
srun -n4 -c2 --exclusive my_app
```

This launches each of the four my_app instances on a separate node, even if the nodes have four or more cores.

--gres=<resource_list>

The first option is used to specify GRES. Each entry in the list has a format of **<name>[[:<type>]:count]**, where name is the name of the resource, count indicates the number of allocated units (one being the default), and type further restricts the resource to a specific class. When used with job steps, using **--gres=none** prevents a specific job step from using any of the resources allocated to the job (by default job steps are permitted to use all GRES allocated to the job). Simultaneous job steps may also partition the job resources by defining their own GRES allocations.

Example

```
srun -N16 --gres=gpu:kepler:2 my_app
srun --gres=help
```

The first example allocates 16 nodes each equipped with two Kepler GPUs for the my_app job. The second invocation may be used to obtain the description of all GRES defined in the specific system.

-C or **--constraint=<features>**

This specifies additional resource constraints that will apply. The option parameter may be a feature name, feature name with associated node count, or an expression formed by concatenating its clauses using the following operators.

- AND ("&"): only the nodes containing all specified features are selected.
- OR ("|"): only the nodes containing at least one of the listed features are chosen.
- Matching OR ("[<feature1>|<feature2>|...]": variant of OR where precisely one of the alternatives is matched.

Currently, jobs steps may only use a single feature name as a constraint (no operators are supported). Features are defined by administrators, and therefore meaningful only on a specific system.

Example

```
srun -n4 -C 'big_mem*2|small_mem*4' my_app
srun -N8 -C '[rack1|rack3|rack5]' my_app
```

The first example reserves two large memory nodes or four nodes with small memory capacity, and starts four user processes on the selection. The second command allocates eight nodes within a single rack selected from three possibilities.

-t or **--time=<time>**

This is one of the most frequently used options, and limits the total runtime of the job allocation. When the execution time limit is reached, all running tasks are sent a TERM signal followed soon thereafter by a KILL signal. Intercepting the first signal may be used to arrange for graceful termination of affected processes. Time resolution is 1 min (seconds are rounded up to the next minute) with allowed specification formats of [<hours>:]<minutes>:<seconds>, <minutes>[:<seconds>], <days>-<hours>[:<minutes>[:<seconds>]]. SLURM is frequently configured to permit a reasonable grace period following the expiration of the job allocation. A time value of zero imposes no temporal limit on the execution.

Example

```
srun -N1 -t15 my_app
srun -N8 -t1-3:30 my_app
```

The first command executes the job for 15 min on one node. The second will allocate eight nodes for 1 day, 3 h, and 30 min.

-i or **--immediate[=<seconds>]**

--begin=<time>

--deadline=<time>

These options additionally affect the temporal aspect of job scheduling. The first attempts to start the job within a specified period given in seconds (resources must be available right away if no argument is present). The job is not started if the resources cannot be allocated within the time indicated. The last two options may be used either to postpone the start of the job until a specific time (**begin**) or to make sure that it finishes before a certain time (**deadline**). The latter removes the job if completion by the deadline is not possible. The time specification format for both is YYYY-MM-DD [THH:MM[:SS]] for each letter standing for year, month, day, hour (24 h clock), minute, and second. Letter "T" separates the date from the time. If launching on the same day, just the time specification may be used without the letter "T" and with optionally appended "AM" or "PM". Both options offer additional time formats for convenience (see the examples).

Example

```
srun -N4 --deadline=5/27-16:30 -t1-0 my_app
srun -N8 --begin="now+300" my_app
srun -N1 --begin=noon my_app
```

The first example sets the completion deadline for an application estimated to run for a single day to May 27th at 4:30 p.m. in the current year. The second command will attempt to schedule the application within the next 5 min after submission (default units are seconds, but "minutes" and "hours" may be specified following the number). Finally, the third example will limit the job start to no later than noon (note that this may be the current or the following day, depending on the time of submission). Other predefined times of day include **midnight**, **teatime** (4 p.m.), and **fika** (3 p.m.).

-d or **--dependency=<list_of_dependencies>**

This defers job execution until the listed dependencies are satisfied. This option applies only to full jobs and not job steps. list_of_dependencies may assume one of two forms, one using commas, ",", to separate the entries, while the other uses question marks, "?". With the first format all specified dependencies must be satisfied for the job to be launched. The other form means that satisfying any of the dependencies is sufficient for the dependent job to be started. Each entry assumes one of the following expressions:

- **after:<id>[:<id>...]** delays the dependent job start until all listed jobs start the execution
- **afterany:<id>[:<id>...]** defers the dependent job until the listed jobs terminate
- **aftercorr:<id>[:<id>...]** is used to start tasks in the current job array after successful completion of the corresponding tasks in the listed job array
- **afternotok:<id>[:<id>...]** specifies dependency on failed jobs (timed out, nonzero exit code, node failure, and others)

- **afterok:<id>[:<id>...]** starts the job after successful completion of listed jobs (completed with zero exit code)
- **expand:<id>** indicates that resources allocated to this job are used to expand the job <id>, which must execute in the same partition
- **singleton** defers the execution of this job until all previously started jobs with the same name and by the same user terminate.

Example

```
srun -N4 --dependency=afterok:1234 my_app
```

This will not start the job involving "my_app" until job 1234 completes successfully.

-J or **--job-name=<name>**

This permits the user to specify the job name. The default is to use the submitted executable name. The job name is displayed alongside the job ID when listing the queue contents.

Example

```
srun -N4 --job-name=gamma_ray_4n my_sim
```

This will change the default job name `my_sim` to `gamma_ray_4n`.

--jobid=<id>

This initiates a job step under an already allocated job with the specified ID. For regular users, this command is limited to job step control only and should not be used for full job allocations.

--checkpoint=<time>

--checkpoint-dir=<path>

--restart-dir=<path>

These handle automatic checkpointing and restart. The first option will create checkpoints at regular intervals specified by the time argument. The time format is identical to that used by the **time** option. The default is not to generate checkpoints. The directory to store the checkpoint data is defined by the second option, defaulting to the current working directory. The third option specifies the directory from which the checkpoint data will be read when restarting a job or job step.

Example

```
srun -N4 -t40:00:00 --checkpoint=120 --checkpoint-dir=/tmp/user036/chckpts my_app
```

This will run the job for 40 h, checkpointing its state every 2 h. The checkpoint files are stored in a user's subdirectory on a temporary file system.

-D or **--chdir=<path>**

This changes the current working directory to the path specified before initiating job execution. The default is the working directory used for job submission. The path may be absolute or relative to the current working directory.

Example

```
srun -N64 -t10:00 -D /scratch/datasets/0015 dataminer.sh
```

This will switch the working directory to scratch storage before starting the application.

-p or **--partition=<partition_name>**

This specifies a partition (queue) to be used. A comma-delimited list of partitions may be specified to accelerate the job allocation.

Example

```
srun -N4 -t30 -p small,medium,large my_app
```

This will start the application in the small, medium, or large job queue, whichever becomes available first.

--mpi=<mpi_type>

This identifies the MPI implementation to use. Supported types (which may not be supported on all systems) include:

- **openmpi** enables the use of OpenMPI library and implementation
- **mvapich** supports MPI implementation on InfiniBand
- **lam** with one *lamd* process per node and appropriate environment variables
- **mpich1_shmem** launches one process per node and environment initialized for shared memory support in either MPICH1 or MVAPICH shared memory build
- **mpichgm** to be used with Myrinet networks
- **pmi2** if the underlying MPI implementation supports the process management interface (PMI2)
- **pmix** includes support for PMI1, PMI2, and PMIx, and requires that SLURM is configured accordingly
- **none** used for other MPI environments.

Example

```
srun -N64 -t300 --mpi=mvapich mpirun my_sim
```

This will run the MPI application `my_sim` on 64 nodes using InfiniBand interconnect.

-l or **--label**

This prepends a task number to every output line (for both stdout and stderr) generated while running the job. Since the output of all processes may be interleaved on the console, this option helps identify and sort the output lines printed by individual tasks. This has uses in debugging and post-mortem analysis of applications.

Example

```
srun -N4 -l hostname
```

A possible output is shown below:

```
1: node06
0: node05
3: node08
2: node07
```

-K or **--kill-on-bad-exit[={0,1}]**

This determines whether to terminate the job if one of its tasks fails (exits with nonzero status). The job will not be terminated if the argument "1" is specified; in all other cases ("0" or no argument) task failure will imply the job's failure.

-W or **--wait=<seconds>**

This specifies the waiting period in seconds for other task termination after completion of the first task. "0" signifies unlimited waiting time, with a warning issued after the first 60 s. The **kill-on-bad-exit** option takes precedence over **wait**, causing the immediate termination of other tasks after the first one exits with nonzero status.

5.2.4.2 salloc

salloc [<options>] [<command> [<command_arguments>]]

The **salloc** command obtains resource allocation and runs the command specified by the user. The allocation is relinquished after the user's command completes. The **salloc** command manipulates terminal settings and therefore should be executed in the foreground. The command may be an arbitrary program or possibly shell script containing **srun** commands. The job output is shown directly on the terminal from which the command was invoked. The resource allocation options are identical to those listed for **srun** above, with the addition of the following:

-F or **--nodefile=<path>**

Similarly to the nodelist option described above, this explicitly specifies names of the nodes to be used for allocation. The names are stored in a file identified by **path** argument. The node names may be listed in multiple lines. Duplicates and ordering do not matter, as the list will be sorted by SLURM.

5.2.4.3 sbatch

sbatch [<options>] [script [<arguments>]]

The **sbatch** command is used to submit batch scripts for execution to the SLURM system. This is the preferred way of running large or long jobs, as it allows the scheduler to pick the right moment for their launch to maintain high system utilization and job throughput. The job parameters are fully described by **sbatch** command line options and script contents, including I/O stream redirection. This frees the user from being continuously present at the terminal. The script may be a file or, if omitted on the command line, entered directly on the terminal. Batch script contents are described in more detail in the next subsection.

Normally, **sbatch** exits as soon as the script is successfully submitted to the SLURM controller daemon. This does not mean that the job has executed, or even that it has been allocated resources, only that it has been queued. When the resources for execution are granted, SLURM starts a copy of the submitted script on the first of the assigned nodes. If commands executed by the script generate any output, it is stored in files with the name "slurm-%j.out", where "%j" is the job number. For job arrays the output is captured in files named "slurm-%A_%a.out", with "%A" denoting job identifier and "% a" job index.

Like **sallocate**, **sbatch** recognizes many of the same resource allocation options, but also supports a few of its own.

-a or **--array=<index_list>**

This submits a job array containing multiple jobs with the same parameters. The **index_list** specifies numerical IDs of individual jobs and may use comma-delimited numbers, ranges (two numbers separated by a dash), and step functions (range followed by a colon and a number). Additionally, the user may put a restriction on a number of simultaneously executing tasks from the job array by suffixing the index_list with a "%" (percent sign) and a number.

Example

```
sbatch -N6 -a5-8,10,15%3 script.sh
sbatch -N2 -a0-11:5 script.sh
```

This will create a six-job array with job indexes 5, 6, 7, 8, 10, and 15 while limiting the number of concurrent tasks to three. The second command creates a job array with three jobs indexed 0, 5, and 10.

-o or **--output=<pattern>**

This redefines the default file name to store the job script's output stream with a **pattern**. The pattern may be an arbitrary literal that could be used as a file name by the underlying file system with special character sequences that are expanded by SLURM using current job parameters. They include:

- \\ to suppress the processing of expansion sequences
- **%%** to insert the single "%" character
- **%A** expands into the job array's master job allocation number
- **%a** produces a job index within a job array
- **%j** yields a job allocation number
- **%N** is the node name of the first node used by the allocation
- **%u** converts to the user's name.

Example

```
sbatch -N10 -o"ljs-%u-%j.out" ljs.sh
```

This will capture the job's output in file "ljs-joe013-1337.out" if submitted by user joe013 and the allocated job number was 1337.

-W or **--wait**

This postpones the **sbatch** exit until the submitted job terminates. The exit code of **sbatch** will be the same as the exit code of the job, and for job arrays it will be the highest recorded exit code of all jobs in the array.

5.2.4.4 squeue

squeue [<options>]

The **squeue** command displays information about jobs and job steps in SLURM queues. It may be used to examine the status of queued, running, and suspended jobs, and show their resource allocations, time limits, associated partitions, and job owners. The frequently used options are as follows.

--all

-l or **--long**

These force additional information to be shown. The **all** option displays the status of jobs in all partitions, including hidden partitions and partitions that are unavailable to the user invoking the command. The **long** option is specified to list the contents of additional fields, e.g., time limit for each job.

-M or **--clusters=<cluster_list>**

-p or **--partition=<partition_list>**

-u or **--user=<user_list>**

-t or **--states=<state_list>**

These restrict the reported information to specific clusters, partitions, users, or states. Each option accepts a single name or comma-separated list of applicable names (for the first three options they are system dependent). The **states** option accepts the following state IDs, listed here in full and shortened format: PENDING (PD), RUNNING (R), SUSPENDED (S), STOPPED (ST), COMPLETING (CG), COMPLETED (CD), CONFIGURING (CF), CANCELLED (CA), FAILED (F), TIMEOUT (TO), PREEMPTED (PR), BOO_TFAIL (BF), NODE_FAIL (NF), and SPECIAL_EXIT (SE). The state IDs are case insensitive.

Example

```
squeue -presearch -tPD,S -i60
```

This lists all pending and suspended jobs for the research partition of the currently used cluster, and updates it every minute.

-i or **--iterate=<seconds>**

This repeatedly updates the displayed information every given number of seconds. The time stamp of the last update is included in the header.

--start

This shows the expected start time and resource allocation for pending jobs if the SLURM scheduler is configured with the backfill plugin. The output is ordered by increasing start time.

-r or **--array**

This prints each job element per line when showing job arrays. If not specified, the output contains condensed information about job arrays combining all information about each job array into a single line.

5.2.4.5 scancel

scancel [<options>] [<job_id>[_<array_id>][.<step_id>]]...

The **scancel** command cancels or delivers signals to jobs, job arrays, and job steps. Besides the options, **scancel** accepts any number of arguments denoting the identifiers of specific jobs or job steps. An underscore ("_") is used to specify the individual elements of a job array. Both regular jobs and job array elements may append a step identifier after a period (".") to limit the scope of signal delivery to the specific job steps. The target job subset may also be identified by application of filters, in which case no explicit job identifiers need be given.

The essential command options include the following.

-s or **--signal=<signal>**

This determines the type of Unix signal to be delivered. The **signal** argument may be either the signal's name or its number, and is typically one of HUP, INT, QUIT, ABRT, KILL, ALRM, TERM, USR1, USR2, CONT, STOP, TSTP, TTIN, and TTOU. Absence of this option causes job termination.

Example

```
scancel -sSTOP 12345
```

This will send the STOP signal to job number 12345.

-n or **--name=<job_name>**

-p or **--partition=<partition_name>**

-t or **--state=<job_state>**

-u or **--user=<user_name>**

These options restrict the set of jobs affected by **scancel**. The job filtering may be done by job name, partition name, state, or user ID of the job's owner, respectively. The job state must be PENDING, RUNNING, or SUSPENDED.

Example

```
scancel -tPENDING -ujoe013
```

This terminates all pending jobs owned by user "joe013".

-i or **--interactive**

This enables an interactive mode in which the user has to confirm the cancellation of each affected job.

5.2.4.6 sacct

sacct [<options>]

This retrieves job accounting data from SLURM logs or databases. Information is collected on jobs, job steps, their status, and exit codes. This command may also be used to access the status of no longer existing jobs to determine if they completed successfully. The options available to the regular user include the following:

-a or **--allusers**

-L or **--allclusters**

-l or **--long**

-D or **--duplicates**

The options listed above increase the amount of information reported by **sacct**. The first outputs data related to jobs owned by all users of the cluster (note that this may be restricted in some environments). Similarly, **allclusters** includes data collected for all clusters under SLURM control; otherwise the output is limited to the machine from which the command is invoked. The **long** option provides practically all information that has been retained in logs pertaining to the finished job. Finally, the last option provides information for all jobs that used the same ID. Normally, only the records with the most recent timestamp are reported for each job ID.

-b or **--brief**

-j or **--job=<job>[.<step>]**

--name=<jobname_list>

-s or **--state_list=<state_list>**

-i or **--nnodes=<min_nodes>[-<max_nodes>]**

-k or **--timelimit-min=<time>**

-K or **--timelimit-max=<time>**

-S or **--startime=<time>**

-E or **--endtime=<time>**

Options in this group filter or otherwise restrict the output of the **sacct** command. The **brief** option shortens the listing to just job ID, status, and exit code. The **job** and **name** take arguments that identify the specific job (or job steps) and job names of interest. The **state_list** will list jobs that are pending, executing, or terminated in a specific state. The state mnemonics include (short form in parentheses) CANCELED (CA), COMPLETED (CD), COMPLETING (CG), CONFIGURING (CF), PENDING (PD), PREEMPTED (PR), RUNNING (R), SUSPENDED (S), RESIZING (RS), TIMEOUT (TO), DEADLINE (DL), FAILED (F), NODE_FAIL (NF), and BOOT_FAIL (BF). The **nnodes** option shows only entries that allocated a specific number of nodes (a range may be specified). The remaining options are used to limit the retrieved records by the range of execution time limits (**timelimit-max** may only be specified if **timelimit-min** is set), and actual start and end times. The time format is the same as for the **srun time** option.

Example

```
sacct -sF,NF,BF -a -D
```

This will list all failed jobs (including errors due to node failures) on the current machine.

5.2.4.7 *sinfo*

sinfo [<options>]

This shows information about system partitions and nodes managed by SLURM. The options **all**, **long**, **clusters**, **partition**, and **iterate** are available, and have the same semantics as described above for **squeue**. In addition to these, **sinfo** interprets the following options.

-n or **--nodes=<node_list>**

This displays information only about the specified nodes. Node names may be individually listed in a comma-separated list or use range syntax, as described for the **nodelist** option of **srun**.

-r or **--responding**

-d or **--dead**

command line (executable with options and arguments) for each application used. The program arguments in the configuration file may contain percent sign ("%") expressions that will be replaced by relevant job parameters when actually run:

- **%t** is replaced by the task number under which the application executes
- **%o** expands to task offset within a range specified at the start of the line for the application.

For example, we can create the file "multi.cf" with the following contents:

```
2,7    hostname
0-1,6  echo sample task A: task=%t offset=%o
3-5    echo sample task B: task=%t offset=%o
```

We use the script shown below to execute the job:

```
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4
srun -l --multi-prog multi.cf
```

Option **ntasks-per-node** forces the distribution of tasks across two nodes, while the **–l** option passed to **srun** causes it to prefix every output line with the number of the task that prints it out. Script execution produces the following output:

```
0: sample task A: task=0 offset=0
1: sample task A: task=1 offset=1
3: sample task B: task=3 offset=0
2: node02
5: sample task B: task=5 offset=2
6: sample task A: task=6 offset=2
7: node03
4: sample task B: task=4 offset=1
```

The second method to achieve concurrent execution of different applications is by spawning simultaneous job steps. The following script illustrates the concept:

```
#!/bin/bash
#SBATCH --ntasks=1536
#SBATCH --time=1:00:00
srun -n1 ./single_process &
srun -n16 mpirun ./small_mpi_app &
srun -n1024 mpirun ./big_mpi_app &
wait
```

The concurrent job steps are created by placing an ampersand ("&") at the end of the relevant lines, which cause the **srun** command to execute in the background. Note that unlike **multi-prog**, this method enables concurrent execution of parallel applications. The `wait` statement is required to prevent script exit before all the background job steps complete. On systems with installed PMI, the `mpirun` commands may be dropped, since the MPI applications already include support for parallel launch. Also, the resource requests in the script header need not exactly match the aggregate resource allocations of all simultaneous job steps. However, if they are significantly overestimated the unnecessarily increased amount of requested resources may delay job execution. As a general rule, creating multiple job steps is preferable to submitting multiple jobs, as the mechanisms used to launch the job steps introduce much lower overheads than full-scale job resource allocation and scheduling.

5.2.5.5 Environment Variables

The execution of scripts may be further modified by using environment variables that are provided by SLURM to reflect the details of resource assignment to a particular job and expose information that is not known prior to its execution. These environment variables (only a subset is shown) may be categorized in the following groups.

- Propagated option values

  SLURM_NTASKS or **SLURM_NPROCS**

  SLURM_NTASKS_PER_CORE

  SLURM_NTASKS_PER_NODE

  SLURM_NTASKS_PER_SOCKET

  SLURM_CPUS_PER_TASK

  SLURM_DISTRIBUTION

  SLURM_JOB_DEPENDENCY

  SLURM_CHECKPOINT_IMAGE_DIR
  These variables reflect the values of **sbatch** options specified either on the **sbatch** command line or in the job script header. They correspond respectively to the **ntasks**, **ntasks-per-core**, **ntasks-per-node**, **ntasks-per-socket**, **cpus-per-task**, **distribution**, **dependency**, and **checkpoint-dir** options.

- Counts of resources allocated to the job:

  SLURM_JOB_NUM_NODES or **SLURM_NNODES** holds the total number of nodes allocated to the job

  SLURM_JOB_CPUS_PER_NODE, depending on the scheduler, indicates the total number of CPUs (cores) available on the local node or the actual number of CPUs allocated to the job

  SLURM_CPUS_ON_NODE indicates the number of CPUs on the current node.

- Runtime assigned IDs and enumerations:

  SLURM_SUBMIT_HOST specifies the name of the host on which the job was submitted

  SLURM_CLUSTER_NAME contains the name of the cluster on which the job is running

  SLURM_JOB_PARTITION names the partition in which the job is running

  SLURM_JOB_ID or **SLURM_JOBID** indicates the ID of the current job

SLURM_LOCALID indicates the ID of the node-local task corresponding to the current process

SLURM_NODEID is the ID of the allocated node

SLURM_PROCID specifies the global relative ID of the current process (MPI rank if the process is a part of MPI process group)

SLURM_JOB_NODELIST or **SLURM_NODELIST** contains a list of node names that were allocated to the job; it may contain node ranges or individual entries

SLURM_TASKS_PER_NODE shows the number of tasks executing on each node; entries in the list correspond to host names in the **SLURM_JOB_NODELIST** variable, with some space-saving notation applied to identical consecutive entries (e.g., 4(x2) indicates two consecutive nodes with a task count of 4)

SLURM_ARRAY_TASK_ID stores the index of job array elements

SLURM_ARRAY_TASK_MIN and **SLURM_ARRAY_TASK_MAX** provide the minimum and maximum indices used by the job array

SLURM_ARRAY_TASK_STEP indicates the step by which the index is increased in the job array

SLURM_ARRAY_JOB_ID specifies the ID of the master job in the job array.

- Other

SLURM_SUBMIT_DIR contains the directory name from which the job was submitted

SLURM_RESTART_COUNT stores the current count if the job has been restarted due to failure or requeueing.

Importing the actual configuration parameters into the script and application space through environment variables permits nearly arbitrary customization of job execution. It also enables creation of more flexible job scripts. For example, the following script calculates the total number of cores allocated to the job and selects the appropriate input configuration based on the outcome. It also provides a unique log file name to be generated by the master task, reflecting the job number and used resource geometry.

```
#!/bin/bash

job=$SLURM_JOB_ID
nodes=$SLURM_JOB_NUM_NODES
cores=$SLURM_JOB_CPUS_PER_NODE
total=$((nodes * cores))

config=small.conf
[ $total -ge 4096 ] && config=medium.conf
[ $total -ge 16384 ] && config=large.conf

mpirun ./my_sim -i $config -o sim_${job}_${nodes}x${cores}.log
```

SLURM environment variables may be helpful in staging files to a higher performance file system than shared Network File System storage (frequently used to provide global access to home directories). The script listed below creates a unique temporary directory for each task in local temporary storage, copies the dataset `data.in` prepared in the submission directory, spawns tasks that modify it, and copies the results back. It assumes that the cluster supports a passwordless secure shell on login and compute nodes.

```
#!/bin/bash

host=$SLURM_SUBMIT_HOST
hostdir=$SLURM_SUBMIT_DIR
tmpdir=/tmp/${USER}/${SLURM_JOB_ID}

srun mkdir -p $tmpdir/$SLURM_PROCID
srun scp ${host}:${hostdir}/data.in \
  $tmpdir/$SLURM_PROCID/data

srun ./update_file $tmpdir/$SLURM_PROCID/data

srun scp $tmpdir/$SLURM_PROCID/data \
  ${host}:$hostdir/data.out.${SLURM_PROCID}
```

5.2.6 SLURM CHEAT SHEET

This subsection contains a collection of commands that accomplish frequently performed tasks but may sometimes be difficult to locate in the manual. They are presented in the way they would be typed by a user at the login shell prompt, although many of them can be converted to the equivalent job scripts. The examples below serve primarily as a template, since in many cases the option arguments are strongly platform dependent. For commands that require resource allocation, both the time limit and the number of nodes or tasks are specified to enforce good practices.

Invoke the interactive shell on the allocation:

```
srun -N4 -t30 --pty /bin/bash
```

Enable X windows forwarding for graphical applications (requires an X11 plugin installed):

```
srun -N1 -t30 --x11 xterm
```

(Here xterm is used as an example application).

Submit job to the specific queue ("debug" in this case):

```
sbatch -N4 -t30 -pdebug job.sh
```

Submit a multithreaded MPI job (MPI with OpenMP). The command below spawns 16 MPI processes with 8 OpenMP threads each, placing 2 processes per node:

```
env OMP_NUM_THREADS=8 sbatch -n16 -c8 -t30 \
--ntasks-per-node=2 job.sh
```

Specify memory requirements for the job (4 GB = 4096 MB per node shown):

```
sbatch --mem=4096 -n2 -t30 job.sh
```

Find out the estimated start of execution time (1234 is the queued job identifier):

```
squeue --start -j 1234
```

Ask to be notified by email when the job terminates or fails:

```
sbatch --mail-type=END,FAIL -N4 -t30 job.sh
```

Kill a submitted or currently running job (1234 is the identifier of the queued job):

```
scancel 1234
```

5.3 THE ESSENTIAL PORTABLE BATCH SYSTEM

5.3.1 PORTABLE BATCH SYSTEM OVERVIEW

PBS is one of the oldest resource management suites. It originated in 1991 as a contract project for NASA, with the bulk of the proprietary code developed by MRJ Technology Solutions. The PBS interface was based on the POSIX 1003.2d standard defining batch environments, which was ultimately released in 1994. The underpinnings of the initial PBS design were the result of collaboration between NASA Ames, Lawrence Livermore National Laboratory, and the National Energy Research Scientific Computing Center. Further developments brought integration with operating environments on Cray (UNICOS), Intel Paragon, and iPSC/860, as well as checkpoint/restart, interactive job support, and initiated experiments resulting in execution of workloads on supercomputers located at opposite ends of the United States (NASA Metacenter). In 1998 the PBS team led by Bill Nitzberg released version 2.0 of the resource manager code, which included the ability to add and remove execution nodes dynamically. Two years later Veridian Corp. announced the first commercial release of PBS,

PBS Pro 5.0. At this point PBS supported an advanced reservation mechanism and peer scheduling, and was capable of managing grid workloads using Globus. The intellectual property behind the proprietary version of PBS has been acquired by Altair Engineering, which remains its home to this day. The improvements that followed include topology-aware scheduling, on-demand computing, scheduling on GPUs, and support for performance data analysis and visualization. In May 2016 Altair opened the code base of PBS Professional to stimulate innovation across all markets important to the HPC community.

Several open-source PBS implementations compatible in essential functionality but not in all of the features have been developed over the years. The most notable are the following.

- **OpenPBS**, deriving from the revision open sourced by the MRJ in the late 1990s. This version is no longer in development.
- **TORQUE**, or Terascale Open-Source Resource and Queue Manager. TORQUE was developed and is supported by Adaptive Computing with significant community contributions. It includes such features as job arrays, GPU scheduling, high-throughput support, advanced diagnostics, log and statistics collection, node health monitoring, and high availability.

Both open and proprietary versions of PBS were extensively used by among others the NASA Goddard Space Flight Center, Chevron, Conoco, Wolfram Research, Nvidia, the US Department of Defense, Department of Energy national laboratories, National Center for Supercomputing Applications, and Australian National Computational Infrastructure. Altair Engineering and Adaptive Computing formed partnerships with Hewlett-Packard, Cray, Silicon Graphics International, Fujitsu, Groupe Bull, and others as resellers of resource management products. Software from both companies was also awarded the Intel "cluster ready" certification. PBS maintains a very strong presence in the HPC community and is one of the most popular and broadly used resource management systems.

5.3.2 PORTABLE BATCH SYSTEM ARCHITECTURE

Similar to SLURM, PBS consists of a number of daemons accepting user commands and sharing job execution duties. User-command processing, job creation, monitoring, and dispatch, and protecting

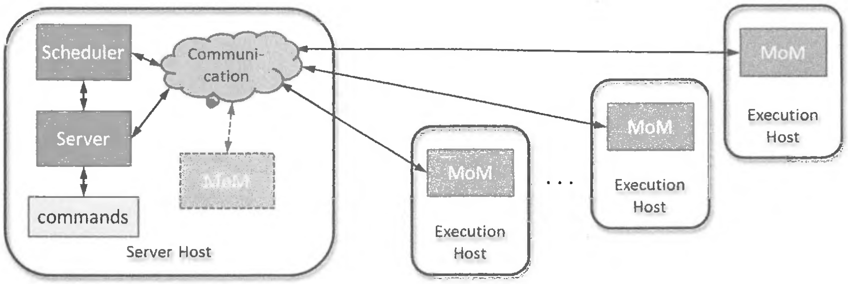


FIGURE 5.3

Simplified PBS architecture.

From Altair/PBS

against system failures are the responsibility of the server daemon (Fig. 5.3). The server runs on the cluster's head node, or server host in PBS terminology, and interacts with other entities in the system via the communication daemon. The communication daemon is based on the internet protocol. There is one server for each set of resources.

The compute nodes (execution hosts in PBS) typically run only the machine-oriented miniserver (MoM) daemons, one instance per node. MoMs play the role of job executors, and are more commonly described as "the mother of all executing jobs". MoMs communicate with the server to receive the jobs to be run on local execution resources. They are also responsible for faithfully instantiating shell-like user sessions, including the correct initialization of the related environment (in particular execution of the appropriate shell initialization scripts and set up of environment variables) as well as proper redirection of I/O and error streams.

The scheduler is responsible for monitoring the state of system resources and deciding when and on which subset of resources each job is to run. It does so by polling the MoM daemons to obtain the most current utilization data. The scheduler also communicates with the server to sample the status of job queues and thus determine the next most eligible jobs to execute.

The configuration described above is most typical for small and medium-scale platforms. Optionally, a PBS installation may include additional server hosts to reduce the amount of resources that need to be managed per server instance. In some cases MoM daemons may be permitted to execute on server hosts to extend the pool of available resources. Finally, PBS allows the inclusion of nodes whose only function is command submission.

5.3.3 SUMMARY OF PBS COMMANDS

5.3.3.1 qsub

qsub [<options>] [<script_name>]

The **qsub** command is likely the most frequently used command in PBS. It allows users to submit jobs to the batch system, along with their resource requirements and additional attributes. If the name of the script file is omitted, **qsub** reads the equivalent statements from the terminal input or starts the application specified on the command line. After the submission is successfully accepted, **qsub** prints the job identifier in the form `<sequence_number>.<server_name>` or, for job array, `<sequence_number>[].<server_name>`. Job parameters can be defined directly in scripts (discussed in Section 5.3.5) or passed through command-line options. These include the following.

-l <resource_list>

This frequently used option requests resources, specifies distribution of job components, and imposes limits on various aspects of job execution. To allocate job-wide resources, the option's argument assumes the format:

<resource_name>=<value>[,<resource_name>=<value>...]

The following are some of the supported resource names.

- **Nodes**—number and type of nodes to be allocated for the job. They are described using the following format:

<node_spec>[+<node_spec>...]

where each **<node_spec>** starts with the number of nodes followed by one or more named properties separated by colons, ":". If no number is provided, "1" is assumed. The properties may be:

- name of the node (hostname)
- **ppn=<processors_per_node>** (defaults to 1)
- another string assigned by the system administrator which may identify additional parameters of interest, such as memory size, CPU type, or accelerator availability.

- **Walltime**—the maximum amount of time a job is permitted to run.
- The names **cput** and **pcput** refer respectively to the aggregate CPU time used by all processes and the maximum time used by any of the job's processes (unit: time).
- The names **pmem**, **pvmem**, and **vmem** are respectively the maximum physical memory used by any of the processes, maximum virtual memory used by any of the processes, and maximum virtual memory used by all processes in aggregate (unit: size).
- **File**—the maximum size of any of the files created by the job (unit: size).

The **qsub** command also offers new style resource selections and job placement statements. The new syntax uses the abstraction of *vnodes* (virtual nodes) to make the resource allocation and partitioning more flexible. Vnodes represent a set of resources that are a usable part of a machine. A vnode can be an entire host or a part of it, such as a single processing blade. A host may comprise multiple vnodes.

The new style resource allocation formats are incompatible with the syntax described above, thus mixing these two approaches in a single job will result in an error. The resource selection is specified using the following format:

select=[<number>:]<chunk>[+[<number>:]<chunk>...]

where **<number>** determines how many instances of **<chunk>** are needed. Each chunk is a list of **<resource_name>=<value>** assignments separated by colons (":"). Some of the commonly used built-in resources include:

- **arch**—type of the architecture (site dependent)
- **ncpus**—number of processing cores
- **mem**—amount of physical memory allocated to the chunk
- **mpiprocs**—number of MPI processes per chunk
- **accelerator**—indicates whether the chunk contains an accelerator
- **naccelerators**—number of accelerators on the host (host-level resource)
- **accelerator_memory**—amount of memory with which accelerators on this vnode are equipped
- **accelerator_model**—type of accelerator associated with a vnode
- **ompthreads**—number of OpenMP threads

- **host**—name of host to execute the job on
- **vnode**—name of virtual node to be used for execution.

The placement format supported by the **-l** option must conform to the following:

place = [<arrangement>][:<sharing>][:<grouping>]

The following rules apply.

- *arrangement* may be one of **free**, **pack**, **scatter**, or **vscatter. free** will place the job on any of the vnodes, **pack** will put all chunks on a single host, **scatter** will assign only one MPI chunk to a host (although nonMPI chunks may be assigned to the same node), and **vscatter** takes one chunk from one vnode.
- *sharing* keywords include **excl**, **shared**, and **exclhost**. They determine the exclusivity of vnode allocation for the job. The first permits only this job to use the allocated vnodes, the second allows vnode sharing, and **exclhost** allocates the entire host to the job.
- *grouping* determines how chunks are grouped according to a resource. It takes the form of **<group> = <resource>** with **<resource>** being either a built-in resource host or a node-level resource that is site specific.

Some of the option variants expect arguments that express time. The time value must conform to the string in the format **[[<hours>:]<minutes>:]<seconds>[.<milliseconds>]**. Other arguments denote size, which is expressed by a number followed by either **b** or **w** for bytes or words, respectively. The actual word size in bytes is system dependent and equal to the native word size on the execution host. The specification permits **k**, **m**, and **g** (kilo, mega, and giga) prefixes that scale the basic unit 2^{10}, 2^{20}, and 2^{30} times, respectively.

Example

```
qsub -l nodes=16 -l walltime=15:00 my_job.sh
```

This will submit a job described by the script `my_job.sh` to be executed on 16 nodes for at most 15 min.

```
qsub -l nodes=node01+node20+node21,walltime=1:00:00 my_job.sh
```

This will execute the job on three specific hosts, named “node01”, “node20”, and “node21”, for up to 1 h.

```
qsub -l select=2:ncpus=4:mem=2gb my_job.sh
```

This will submit the job, requesting allocation of two resource chunks with four cores and 2 GB memory each.

-q <destination>

Depending on the argument, this sends the job to a specific queue, server, or queue at a server. The argument format corresponding to these cases is **<queue>**, **@<server>**, and **<queue>@<server>**.

Example

```
qsub -l nodes=2 -q debug test.sh
```

This will submit a two-node job to the debug queue.

-N <name>

The first option permits the user to associate a name with the job. If omitted, it defaults to the name of the script or `STDIN` if submitted from the standard input on the console.

Example

```
qsub -l nodes=12,walltime=10:00:00 -N hurricane job.sh
```

This will submit a job named "hurricane" to the pending job queue.

-J <range>

This declares a job array. The range argument assumes the form **<x>-<y>[:<z>]**, with *x* being the starting index of the array, *y* being the upper bound on the index value, and *z* the step (index increment) value. By default, the step value is one.

Example

```
qsub -J 5-22:5 -l walltime=25:00 job.sh
```

This will create a job array with element job indices of 5, 10, 15, and 20.

-a <date_time>

This postpones job execution at least until the specified time. The argument format is **[[[[CC]YY]MM]DD]hhmm[.SS]**, where **CC** is the century, **YY** is the year, **MM** is the month, **hh** is the hour in 24 h format, **mm** is the minute, and **SS** is the second. The omitted components of **<date_time>** are extracted from the current date and time as long as they specify a time point in the future. If not, the nearest time in the future matching the specified **<date_time>** string is assumed.

Example

```
qsub -l nodes=1600 -a 151630 big_one.sh
```

If submitted on June 17, this will schedule the job execution for on or after July 15, 4:30 p.m., in the same year.

-W <attribute_name>=<value>[,<attribute_name>=<value>...]

This specifies additional job attributes. Due to limited space, only a subset is presented here.

- **depend=<dependency>[,<dependency>...]**

  where the individual dependencies may be
  - **after:<job_id>[,<job_id>...]** delays the execution of this job until all jobs in the list have started execution
  - **afterok:<job_id>[,<job_id>...]** does not start the job until all jobs in the list have terminated successfully
  - **afternotok:<job_id>[,<job_id>...]** waits until all jobs in the list terminate with errors
  - **afterany:<job_id>[,<job_id>...]** postpones the job start until all jobs in the list terminate with any exit status
  - **before:<job_id>[,<job_id>...]** jobs specified in the list may begin execution only after this jobs starts execution
  - **beforeok:<job_id>[,<job_id>...]** jobs in the list may start execution only after this job's successful termination
  - **beforenotok:<job_id>[,<job_id>...]** jobs in the list may execute after the current job terminates with an error
  - **beforeany:<job_id>[,<job_id>...]** argument jobs may begin execution only after this job terminates
  - **on:<number>** this job may start execution only after <number> of dependencies on other jobs has been satisfied.
- **block=true** causes **qsub** to block until the submitted job terminates. The command returns the job's exit status.
- **run_count=<number>** sets the number of times the job should be run.

  Example

```
qsub -l nodes=1 -W depend=afterok:simulate.cluster.edu \
  postprocess
```

This makes the `postprocess` job dependent on the successful termination of the job `simulate`.

-V

-v <variable_list>

These options control the export of environment variables to the job's environment. The first one forces copying of all environment variables and shell functions from the user login environment in which **qsub** is run. The second uses an explicit list of variables to be exported. The entries on the list are separated by commas and take the form **<variable>** or **<variable>=<value>**. The first form simply names the variable to be exported (such a variable must exist in the login environment in which **qsub** is invoked), while the second defines both the name and the value of the exported variable.

-I

-X

These start an interactive job, causing the stdin, stdout, and stderr streams of the job to be connected to the terminal session in which qsub is running. If a job script is provided, only its PBS directives are processed. Jobs belonging to a job array cannot be interactive.

The second option enables an *interactive* job to open X windows on the user's display.

-e <path>

-o <path>

-j {oe,eo,n}

These affect handling of the output and error streams of the job. The first two options save the contents of respectively error and standard output streams to specified files. If omitted, stderr is captured in the file **<job_name>.e<number>**, while stdout stream is redirected to **<job_name>.o<number>**, where **<number>** is the job's ID. The **<path>** may be expressed as **[<host>:]<path>**. Both relative and absolute paths are permitted; in the first case they are relative to the current working directory during **qsub** invocation.

The third option describes how standard error and output streams are merged. The parameters listed above correspond to both merged into stdout, both merged into stderr, and not merged.

-S <path>[@<host>][,<path>[@<host>]...]

-C <prefix>

These options may be used to modify how job scripts are processed by PBS. The first specifies the path to the shell executable acting as an interpreter for the job script. By default the user's login shell is used. If the host name is not entered, only one shell path may be listed that applies to all execution hosts. The second option specifies the literal to be used as a prefix for PBS directives inside the script, nominally "#PBS".

Example

```
qsub -l nodes=1 -S $PBS_EXEC/bin/pbs_python test.py
```

This will execute a python script on the target host.

5.3.3.2 qdel

qdel [<options>] <job_id> [<job_id>...]

This deletes specified job(s). If used without any options, **qdel** removes any queued, running, or suspended jobs. In such a case, the job history is retained. Job deletion begins by sending the affected processes the SIGTERM signal. Afterwards, if there are still any remaining processes belonging to the job, they are sent SIGKILL. Supported options include the following.

-W force

This deletes the job even if the execution host cannot be reached.

-x

This applies to all jobs in the system, including finished and moved jobs. The related job history is also removed.

Example

```
qdel -x mpi_sparse8.some.host.com
```

This will delete job `mpi_sparse8` from the server `some.host.com` irrespective of its status. The job's history will also be erased.

5.3.3.3 qstat

The **qstat** command displays on the standard output status of jobs, queues, or servers. Each of these functions requires a different set of options and command-line arguments, which are briefly discussed below.

5.3.3.3.1 Job Status Query

qstat [<options>] [{<job_id>,<destination>}...]

These are default job status options, as follows.

-J

This shows the status of job arrays only.

-t

This displays the status of jobs, job arrays, and subjobs. When combined with **-J**, it only shows the subjob status.

-p

This replaces values in the time-use column with completion percentages. For job arrays, it lists the percentage of subjobs completed.

-x

In addition to queued and running jobs, this displays the status of finished and moved jobs.

For alternative job status options, the command argument in this mode may be a job ID, which causes the printed information to be limited to that specific job, or a server name, in which case the information is restricted to jobs managed by that server.

-a

The status of running and queued jobs is reported.

-H

This displays the status of finished and moved jobs.

-i

This shows information about waiting, held, and queued jobs.

-r

This lists running and suspended jobs.

-T

This replaces the `Elap Time` field with the estimated time for queued jobs.

-u <user.[,<user>...]

This shows information about jobs owned by a specific user(s).

Long job status options are available from:

-f

The *full* option lists the job information in long format, including job ID, job attributes (one per line), job submission arguments, the job's executable, and the argument list.

5.3.3.3.2 Queue Status Query

qstat -Q [-f] [<destination>[,<destination>...]]

qstat -q {-G,-M} [<destination>[,<destination>...]]

Queue status may be examined using one of two forms. The first displays the status of specified queues, one queue per line. If the **-f** option is given, the full status of each queue is listed, one attribute per line. The destination argument may be **<queue_name>**, **<ququе_name>@<server>**, or **@<server>** (the last reports on all queues managed by the specified server).

The second form shows queue status in alternate format, one queue per line. The additional options are as follows.

-G

This shows size in gigabytes.

-M

This shows size in megawords (8 bytes per word).

5.3.3.3.3 Server Status Query

qstat -B [-f] [-G] [-M] [<server>[,<server>...]

The arguments of this command must be server names. The meaning of options is analogous to that described above for queue status query.

5.3.3.4 tracejob

tracejob [<options>] <job_id>

This extracts and outputs log information about a specific job. Log data includes server (time when the job was queued or modified), scheduler (circumstances that prevent the job from running), accounting (track of the job entering the queue, starting execution, termination, and deletion), and MoM (what happened to job while it was running) information. Supported options are:

-a

-l

-m

-s

Each of these options suppresses the presentation of the respective class of data, in order: accounting, scheduler, MoM, and server. Note that to retrieve MoM's log, **tracejob** has to be invoked on the node where the examined MoM daemon runs.

-c <number>

-n <day_count>

-f <filter>

These options provide additional filtering of displayed data. The first limits the count of specific messages to the **number** of most recent occurrences. The second accesses only the logs that go back no more than **day_count** days. Finally, the filter option excludes specific events from the printout. The filter argument is any of the keywords listed below, or a number formed by using OR on the flags given in parentheses:

- **error** (0x0001) filters internal errors
- **system** (0x0002) filters system errors
- **admin** (0x0004) filters administrative events
- **job** (0x0008) filters job-related events
- **job_usage** (0x0010) filters job accounting information
- **security** (0x0020) filters security violations
- **sched** (0x0040) filters scheduling events
- **debug** (0x0080) filters common debug messages

- **debug2** (0x0100) filters less common debug messages
- **resv** (0x0200) filters reservation debug messages
- **debug3** (0x0400) filters debug messages less common than **debug2**
- **debug4** (0x0800) filters debug messages less common than **debug3.**

-v

This increases the verbosity of presented information. Using this option will include additional error messages in the output.

Example

```
tracejob -a -n 7 -f 0x84 1234
```

This will display log information related to jobs with IDs of 1234 and collected in the past week. Accounting, administrative, and debugging information will not be included.

5.3.3.5 pbsnodes

pbsnodes [<options>] [<host>[<host>...]]

The **pbsnodes** command is used to examine the status of system hosts. This information is obtained through interaction with the PBS server. The command supports a number of different invocation formats that use different option subsets. Necessarily, only some of them are described below.

-a

This lists all hosts and their attributes. The attributes may include jobs that are currently running on the specific hosts and resources that are used by running jobs. Summary information about all consumable resources across all vnodes is reported for each host.

-H <host>[,<host>...]

This outputs all attributes with nondefault values on all hosts listed and their vnodes.

-j

-S

These change the format of information displayed for each vnode. The first includes job-related fields such as vnode name, vnode state, number of jobs per vnode, running and suspended jobs, and total and free memory, CPUs, MICs, and GPUs per vnode. The second option presents system-oriented information that besides vnode name and state contains the values of OS custom and hardware resources, host name, queue attribute, amount of vnode memory, and the count of CPUs, MICs, and GPUs.

-L

This causes pbsnodes to produce output in long format with no restrictions on column width.

5.3.4 PBS JOB SCRIPTING

PBS job scripts share many similarities with SLURM scripts. In both cases the scripts are executed by an interpreter, which is typically a shell such as bash or csh. They also use prefixed comments in the script header to define job parameters. Each line of the PBS script header needs to begin with a "#PBS" prefix, which is followed by the qsub command option. For example:

```
#!/bin/bash
#PBS -J 0-3
#PBS -l nodes=4
#PBS -l walltime=30:00
/home/user13/my_app
```

This will start four instances of the my_app program (one per host) as a job array with an execution time limit of half an hour. While SLURM relies on the built-in shell mechanisms to start the appropriate interpreter (following "#!" in the first line of the script), in PBS this can be changed explicitly using the **-S** option. In addition, the PBS **-C** option may redefine the directive prefix to something other than "#PBS". This helps to accommodate scripts and shells in which comments do not start with the "#" character.

Despite many similarities between PBS and SLURM, there are some noteworthy differences in the default setup of the execution environment. While SLURM exports all environment variables set in a user's login shell to a job's environment, by default PBS does not export anything. The user must therefore use **-v** or **-V** options to control explicitly what is copied to the target job's environment. While SLURM attempts to emulate running in the current directory as much as possible (one may use file paths relative to the submission directory and the captured standard outputs are also placed there), PBS runs in a spool directory. Finally, PBS does not merge the standard output and error streams by default.

5.3.4.1 OpenMP Jobs

In contrast to uniprocessor jobs, OpenMP scripts must explicitly request the number of cores required to support the multithreaded application. This is illustrated below:

```
#!/bin/bash
#PBS -l nodes=1:ppn=16
#PBS -l walltime=45:00
export OMP_NUM_THREADS=16
./my_omp_prog
```

The above script allocates one node with 16 cores for the job. Note that the OMP_NUM_THREADS is in this case set explicitly in the script, but it may also be defined using the **-v** option on the command line. The equivalent script using the new style of "chunked" resource requests is presented below:

```
#!/bin/bash
#PBS -l select=1:ncpus=16:ompthreads=16
#PBS -l walltime=45:00
./my_omp_prog
```

Note that since the **ompthreads** directive automatically sets the OMP_NUM_THREADS environment variable, it is no longer necessary to export it explicitly.

5.3.4.2 MPI Jobs

The basic MPI job script is shown below:

```
#!/bin/bash
#PBS -l nodes=16:ppn=8
#PBS -l walltime=1:00:00
module load openmpi
mpirun mpi_app mpi_app_arg
```

It will allocate 16 execution hosts for the job, scheduling eight single-threaded MPI processes on each of them for a total of 128 parallel processes. Expressing the same using the new syntax yields:

```
#!/bin/bash
#PBS -l select=16:ncpus=8
#PBS -l walltime=1:00:00
module load openmpi
mpirun mpi_app mpi_app_arg
```

As PBS does not automatically export the user's environment to the job, the note in the SLURM section about properly setting up the MPI environment is particularly important here. While SLURM users who forget to do this may be "saved" in some cases by SLURM's automatic propagation of the environment, in PBS the task has to be performed explicitly. In the scripts above, this is ensured by the `module` statements. In general, however, the specific command for the task is platform dependent and should be checked with the system administrator or online manuals.

5.3.4.3 Environment Variables of Interest

- **PBS_ENVIRONMENT**—either PBS_BATCH or PBS_INTERACTIVE.
- **PBS_NODEFILE**—name of the file containing the assigned execution vnodes.

- **NCPUS**—number of usable threads per vnode.
- **PBS_TASKNUM**—the process number on this vnode.
- **PBS_ARRAY_INDEX**—index of subjobs in the job array.
- **PBS_ARRAY_ID**—identifier of the job array.
- **PBS_JOB_ID**—job or subjob identifier; if the latter, the <job_id>[<index>].<server> format is used
- **PBS_JOBNAME**—user-defined job name
- **PBS_QUEUE**—queue from which the job is executed
- **PBS_SERVER**—default submission server.
- **PBS_JOBDIR**—the staging and execution directory for the job.
- **PBS_TMPDIR**—a job-specific temporary directory.
- **PBS_O_WORKDIR**—the absolute path to the job submission directory.
- **PBS_O_HOME**—the value of the HOME variable from the submission environment.
- **PBS_O_HOST**—host name of the machine on which **qsub** was invoked.
- **PBS_O_SHELL**—value of the SHELL variable from the submission environment.
- **PBS_O_PATH**—value of the PATH variable from the submission environment.

5.3.5 PBS CHEAT SHEET

Invoke the interactive shell on the allocation:

```
qsub -I -l nodes=4,walltime=30:00
```

Enable X windows forwarding for graphical applications:

```
qsub -X -I -l nodes=4,walltime=30:00 xterm
```

(Here xterm is used as an example application.)
Submit the job to the specific queue ("debug" in this case):

```
qsub -q debug -l nodes=4,walltime=30:00 job.sh
```

Submit the multithreaded MPI job (MPI with OpenMP). The command below spawns 16 MPI processes with eight threads each:

```
qsub -l select=16:ncpus=8:mpiprocs=1:ompthreads=8 \
  --walltime=30:00 job.sh
```

Specify the memory requirements for the job (4 GB per vnode shown):

```
qsub -l select=2:mem=4gb,walltime=30:00 job.sh
```

Find out the estimated start of execution time (1234.host.org is the queued job identifier):

```
qstat -T 1234.host.org
```

Ask to be notified by email when the job gets aborted or terminates:

```
qsub -m ae -l nodes=4,walltime=30:00 job.sh
```

Kill a submitted or currently running job (1234.host.org):

```
qdel 1234.host.org
```

5.4 SUMMARY AND OUTCOMES OF CHAPTER 5

- Resource management tools are an inherent part of the HPC software stack and perform three principal functions: resource allocation, workload scheduling, and support for distributed workload execution and monitoring.
- Resource allocation takes care of assigning physical hardware, which may range from a fraction of the machine to the entire system, to specific user tasks based on their requirements.
- Resource managers typically recognize the resource types of compute nodes, processor cores, interconnects, permanent storage and I/O devices, and accelerators.
- Resource managers allocate the available computing resources to jobs specified by users.
- Jobs may be executed interactively or batch processed. Batch processing requires all necessary parameters and inputs for job execution to be specified before it is launched.

- Jobs may be monolithic or subdivided into a number of smaller steps or tasks. Each such task is associated with the launch of a specific application program.
- Pending computing jobs are stored in job queues, which define the order in which jobs are selected by the resource manager for execution.
- Most systems use multiple job queues, each with a specific purpose and set of scheduling constraints.
- Common parameters that affect job scheduling include availability of execution and auxiliary resources, priority, resources allocated to the user, maximum number of jobs, requested execution time, elapsed execution time, job dependencies, event occurrence, operator availability, and software license availability.
- Job launchers employ hierarchical mechanisms to alleviate bandwidth requirements and exploit network topology to minimize the amount of data transferred and overall launch time.
- Resource managers must be able to terminate any job that exceeds its execution time or other resource limits, irrespective of its current processing status.
- The software commonly used today includes SLURM, PBS, OpenLava, Moab Cluster Suite, LoadLeveler, Univa Grid Engine, HTCondor, OAR, and YARN.
- There is no common standard specifying the command format, language, and configuration of resource management.
- SLURM is an open-source, modular, extensible, scalable resource manager and workload scheduling software for clusters and supercomputers running Linux or other Unix-compatible OSs.
- SLURM scales to the largest systems in use today, including the fastest supercomputer of 2016, the Sunway TaihuLight, with its 40,000 CPUs (over 10 million cores). It is also used on 5 of the top 10 machines. It can handle up to 1000 job submissions and 500 job executions per second.
- Single points of failure are eliminated through the use of multiple backup daemons, permitting the affected applications to continue running and requesting resources to replace those that fail.
- Job sizes are not necessarily fixed over their execution time; they may grow or shrink, but should not exceed the maximum specified size and time limits. Sophisticated scheduling algorithms are available, including elastic scheduling, gang scheduling, and preemption.
- SLURM integrates support for execution on heterogeneous components, such as GPUs, MIC processors, and other accelerators.
- Gang scheduling supports a scheduling approach in which two or more jobs with similar characteristics are allocated the same set of resources. These jobs are then executed in an alternating fashion, so that only one of them obtains exclusive access to the resources at a time.
- PBS is a resource management suite that is among the most widely employed within HPC, including open-source versions such as OpenPBS and TORQUE.
- PBS consists of a number of daemons accepting user commands and sharing the job execution duties. User-command processing and job creation, monitoring dispatch, and protecting against system failures are the responsibility of the server daemon.
- The PBS server runs on the cluster's head node, or server host in PBS terminology, and interacts with other entities in the system via the communication daemon. Communication is based on the internet protocol, with one server for each set of resources.
- The compute nodes run typically only MoM daemons, with one instance per node.
- The PBS MoMs play the role of job executors.

- MoMs communicate with the server to receive the jobs to be run on local execution resources.
- The scheduler is responsible for monitoring the state of the system resources and deciding when and on which subset of resources each job is to run.

5.5 QUESTIONS AND PROBLEMS

1. Describe the role of resource management systems. Can they be implemented as a part of a conventional OS? Elaborate.
2. Based on Figs. 5.1 and 5.3, what are the primary software components of a resource management system in a cluster? Which physical system components do they rely on?
3. What are the two primary types of jobs? Why are they needed?
4. What are the differences between a job array and job step in SLURM?
5. Imagine you are a system administrator for a newly installed computer composed of 260 nodes in total. Of those, 64 come equipped with GPUs and 36 have substantially larger memory capacity. Your users execute both regular and (infrequently) high-priority jobs. The latter require exclusive access to nonaccelerated hardware resources, but never occupy more than 128 nodes.
   a. Propose a partitioning scheme (enumerate the types and sizes of SLURM partitions) that provides good utilization of the entire machine. Identify any partition overlaps.
   b. What kind of provisions would you implement to facilitate parallel job debugging?
   c. Which SLURM features would you take advantage of to minimize the impact of conflicts between jobs of different priorities?
6. What is backfill? How does it affect computer utilization?
7. Provide a couple of realistic cases that would utilize job dependencies in batch processing. Why would emulating this functionality with blocking statements inside job scripts be ill advised?
8. Write a SLURM command line to schedule an MPI application "mpi_compute" that takes input file argument "my_file.dat" stored in the user's home directory. The application must run on 10,240 cores on a machine equipped with 16-core compute nodes that are available in the "production" partition. The anticipated execution time is 1.5 hours. Also provide an equivalent job script with a correctly formed header.
9. Provide the PBS equivalent of the command described in Question 8.
10. List notable user interface differences between SLURM and PBS. How would you instruct a novice SLURM user with experience in PBS to make her/his initial interactions with the job manager more productive?
11. The following PBS job script was submitted on a machine equipped with dual eight-core CPUs per node:

```
#!/bin/bash
#PBS -N sim3-grid25x4
#PBS -l select=4:ncpus=4:ompthreads=4
#PBS -l place=pack
#PBS -l walltime=2:30:00
#PBS -o ${PBS_O_WORKDIR}/${PBS_JOB_ID}.out
#PBS -j oe
mpirun sim3 -x 25 -y 4
```

What information can be inferred about the scheduled job? How are the application's processes and threads distributed across the physical execution resources? How is the distribution going to change if the fourth line is replaced with:

```
#PBS -l place=scatter
```

REFERENCES

[1] XSEDE: Extreme Science and Discovery Environment, 2011 [Online]. Available: https://www.xsede.org.

[2] SchedMD, Slurm Workload Manager Version 17.02, November 2, 2016 [Online]. Available: https://slurm.schedmd.com.

[3] Altair Engineering, Inc., PBS Professional Open Source Project, 2016 [Online]. Available: http://www.pbspro.org.

[4] OpenLava: Open Source Workload Management, 2011–2015 [Online]. Available: http://www.openlava.org.

[5] Adaptive Computing, Inc., MOAB HPC Suite, 2017 [Online]. Available: http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/.

[6] IBM, IBM DeveloperWorks: Tivoli Workload Scheduler, [Online]. Available: https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Tivoli%20Documentation%20Central/page/Tivoli%20Workload%20Scheduler.

[7] Univa, Grid Engine, 2017 [Online]. Available: http://www.univa.com/products/.

[8] University of Wisconsin-Madison, HTCondor High Throughput Computing, April 23, 2017 [Online]. Available: https://research.cs.wisc.edu/htcondor/.

[9] OAR Home Page, February 25, 2016 [Online]. Available: http://oar.imag.fr.

[10] A. Murthy, Apache Hadoop YARN – Concepts and Applications, August 15, 2012 [Online]. Available: https://hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/.

[11] Top 500. The List, 1993–2016 [Online]. Available: https://www.top500.org.

CHAPTER

SYMMETRIC MULTIPROCESSOR ARCHITECTURE 6

CHAPTER OUTLINE

6.1 Introduction 191
6.2 Architecture Overview 192
6.3 Amdahl's Law Plus 196
6.4 Processor Core Architecture 199
6.4.1 Execution Pipeline 200
6.4.2 Instruction-Level Parallelism 201
6.4.3 Branch Prediction 201
6.4.4 Forwarding 202
6.4.5 Reservation Stations 202
6.4.6 Multithreading 203
6.5 Memory Hierarchy 204
6.5.1 Data Reuse and Locality 204
6.5.2 Memory Hierarchy 205
6.5.3 Memory System Performance 207
6.6 PCI Bus 209
6.7 External I/O Interfaces 213
6.7.1 Network Interface Controllers 213
6.7.1.1 Ethernet 213
6.7.1.2 InfiniBand 215
6.7.2 Serial Advanced Technology Attachment 215
6.7.3 JTAG 218
6.7.4 Universal Serial Bus 220
6.8 Summary and Outcomes of Chapter 6 222
6.9 Questions and Exercises 223
References 224

6.1 INTRODUCTION

The most widely used form of high performance computer is the symmetric multiprocessor (SMP). It represents a class of parallel architectures that exploits multiple processor cores to increase performance through parallelism while maintaining a single image of common memory across the entire

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00006-X

Table 6.1 Some Examples of SMPs and Their Characteristics

Vendor and Name	Processor	Number of Cores	Cores per Central Processing Unit	Memory Capacity	PCIe Slots	Storage Slots
IBM S822LC	IBM POWER8 2.92 GHz	20	10	256 GB	2 × 16-lane Gen.3 3 × 8-lane Gen.3	12 LFF
HPE rx2800 i6	Intel Itanium 9760 2.66 GHz	16	8	384 GB	3 × 16-lane Gen.2 2 × 8-lane Gen.2	8 SFF
Dell PowerEdge R930	Intel E7-8870v4 2.1 GHz	80	20	12 TB	10 Gen.3	24 SFF 8 NVMe
Oracle SPARC T7-4	SPARC M7 4.13 GHz	128	32	4 TB	8 × 16-lane Gen.3 8 × 8-lane Gen.3	8 SFF
HPE ProLiant DL385p Gen8	AMD Opteron 6373 2.3 GHz	32	16	384 GB	3 × 16-lane Gen.2 3 × 8-lane Gen.2	8 SFF

parallel computer. This global virtual address space shared by all of the incorporated processors minimizes the changes from a single processor machine thus simplifying the transformation from sequential applications to parallel programs. SMPs are also referred to as *shared-memory* machines or *cache-coherent* computers. The "S" in SMP stands for symmetric, which refers to the property of equal access times by any processor core to any of the main memory banks. As will be seen, this is at best an approximation as secondary effects cause some variability in load/store operations. But overall SMPs provide balanced operations where data placement need not be a major consideration. This differs from the distributed-memory systems discussed in detail elsewhere in the text.

The principal strength of the SMP architecture family is that it is tightly coupled, i.e., all the components are close together in terms of time–distance of operations, data manipulation, and communication. Some examples of SMPs and their characteristics are provided in Table 6.1.

6.2 ARCHITECTURE OVERVIEW

An SMP is a full-standing self-sufficient computer system with all subsystems and components needed to serve the requirements and support actions necessary to conduct the computation of an application. It can be employed independently for user applications cast as shared-memory multiple-threaded programs or as one of many equivalent subsystems integrated to form a scalable distributed-memory massively parallel processor (MPP) or commodity cluster. It can also operate as a throughput computer supporting multiprogramming of concurrent independent jobs or as a platform for multiprocess message passing jobs, even though the interprocess data exchange is achieved through shared memory transparent to the parallel programming interface. The following sections describe the key subsystems in some detail to convey how they contribute to achieving performance, principally through parallelism and diverse functionality with distinct technologies. This section begins with a brief overview of the full organization of an SMP architecture and the basic purposes of its major components, to provide a context for the later detailed discussions.

Like any general-purpose computer, an SMP serves a key set of functions on behalf of the user application, either directly in hardware or indirectly through the supporting operating system. These are typically:

- instruction issue and operation functions through the processor core
- program instruction storage and application data storage upon which the processor cores operate
- mass and persistent storage to hold all information required over long periods of time
- internal data movement communication paths and control to transfer intermediate values between subsystems and components within the SMP
- input/output (I/O) interfaces to external devices outside the SMP, including other mass storage, computing systems, interconnection networks, and user interfaces, and
- control logic and subsystems to manage SMP operation and coordination among processing, memory, internal data paths, and external communication channels.

The SMP processor cores perform the primary execution functions for the application programs. While these devices incorporate substantial complexity of design (described later), their principal operation is to identify the next instruction in memory to execute, read that instruction into a special instruction register, and decode the binary instruction coding to determine the purpose of the operation and the sequence of hardware signals to be generated to control the execution. The instruction is issued to the pipelined execution unit, and with its related data it proceeds through a sequence of micro-operations to determine a final result. Usually the initial and resulting data is acquired from and deposited to special storage elements called registers: very high-speed (high bandwidth, low latency) latches that hold temporary values. Somewhat simplistically, there are five classes of operations that make up the overall functionality of the processor core.

1. The basic register-to-register integer, logic, and character operations.
2. Floating-point operations on real values.
3. Conditional branch operations to control the sequence of operations performed dependent on intermediate data values (usually Boolean).
4. Memory access operations to move data to and from registers and the main memory system.
5. Actions that initiate control of data through external I/O channels, including transfer to mass storage.

Until 2005 essentially all processors in the age of very large-scale integration (VLSI) technology were single-microprocessor integrated circuits. But with the progress of semiconductor technology reflecting Moore's law and the limitations of instruction-level parallelism (ILP) and clock rates due to power constraints, multicore processors (or sockets) starting with dual-core sockets have dominated the processor market over the last decade. Today processors may comprise a few cores, 6—16, with new classes of lightweight architectures permitting sockets of greater than 60 cores on a chip. An SMP may incorporate one or more such sockets to provide its processing capability (Fig. 6.1). Peak performance of an SMP is approximated by the product of the number of sockets, the number of cores per socket, the number of operations per instruction, and the clock rate that usually determines the instruction issue rate. This is summarized in Eq. (6.1).

$$P_{peak} \sim N_{sockets} * N_{cores\ per\ socket} * R_{clock} * N_{operations\ per\ instruction} \tag{6.1}$$

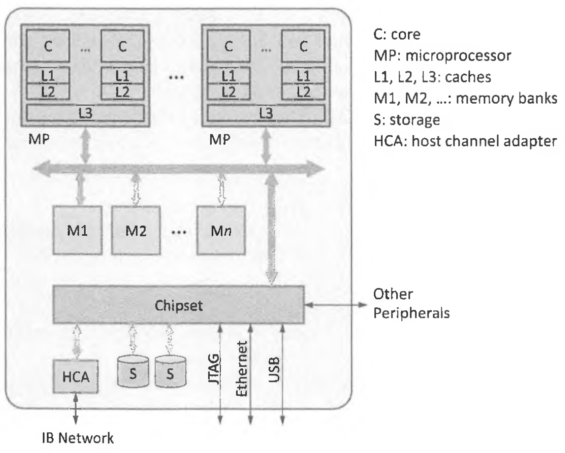


FIGURE 6.1

Internal to the SMP are the intranode data paths, standard interfaces, and motherboard control elements.

The SMP memory consists of multiple layers of semiconductor storage with complex control logic to manage the access of data from the memory by the processor cores, transparent vertical migration through the cache hierarchy, and cache consistency across the many cache stacks supporting the processor core and processor stack caches. The SMP memory in terms of the location of data that is being operated on is, in fact, three separate kinds of hardware. Already mentioned are the processor core registers; very fast latches that have their own namespace and provide the fastest access time (less than one cycle) and lowest latency. Each core has its own sets of registers that are unique to it and separated from all others. The main memory of the SMP is a large set of memory modules divided into memory banks that are accessible by all the processors and their cores. Main memory is implemented on separate dynamic random access memory (DRAM) chips and plugged into the SMP motherboard's industry-standard memory interfaces (physical, logical, and electrical). Data in the main memory is accessed through a virtual address that the processor translates to a physical address location in the main memory. Typically an SMP will have from 1–4 gigabytes of main memory capacity per processor core.

Between the processor core register sets and the SMP main memory banks are the caches. Caches bridge the gap of speeds between the rate at which the processor core accesses data and the rate at which the DRAM can provide it. The difference between these two is easily two orders of magnitude, with a core fetch rate in the order of two accesses per nanosecond and the memory cycle time in the order of 100 ns. To achieve this, the cache layers exploit temporal and spatial locality. In simple terms, this means that the cache system relies on data reuse. Ideally, data access requests will be satisfied with data present in the level 1 (L1) cache that operates at a throughput equivalent to the demand rate of a processor core and a latency of one to four cycles. This assumes that the sought-after data has already been acquired before (temporal locality) or that it is very near data already accessed (spatial locality). Under these conditions, a processor core could operate very near its peak performance capability. But due to size and power requirements, L1 caches (both data and instruction) are relatively small and susceptible to overflow; there is a need for more data than can be held in the L1 cache alone. To address this, a level 2 (L2) cache is almost always incorporated, again on the processor socket for each core or sometimes shared among cores. The L2 cache holds both data and instructions and is much larger than the L1 caches, although much slower. L1 and L2 caches are implemented with static random access memory (SRAM) circuit design. As the separation between core clock rates and main memory cycle times grew, a third level of cache, L3, was included, although these were usually implemented as a DRAM chip integrated within the same multi-chip module packaging of the processor socket. The L3 cache will often be shared among two or more cores on the processor package.

This contributes to achieving the second critical property of the SMP memory hierarchy: cache coherency. The symmetric multiprocessing attribute requires copies of main memory data values that are held in caches for fast access to be consistent. When two or more copies of a value with a virtual address are in distinct physical caches, a change to the value of one of those copies must be reflected in the values of all others. Sometimes the actual value may be changed to the updated value, although more frequently the other copies are merely invalidated so an obsolete value is not read and used. There are many hardware protocols that ensure the correctness of data copies, started as early as the 1980s with the modified exclusive shared invalid [1] family of protocols. The necessity to maintain such data coherence across caches within an SMP adds design complexity, time to access data, and increased energy.

Many SMP systems incorporate their own secondary storage to hold large quantities of information, both program codes and user data, and do so in a persistent manner so as to not lose stored information after the associated applications finish, other users employ the system, or the system is powered down. Mass storage has usually been achieved through hard magnetic disk technology with one or more spinning disk drives. More recently, although with somewhat lower density, solid-state drives (SSDs) have served this purpose. While more expensive, SSDs exhibit superior access and cycle times and better reliability as they have no moving parts. Mass storage presents two logic interfaces to the user. Explicitly, it supports the file system consisting of a graph structure of directories, each holding other directories and end-user files of data and programs. A complete set of specific file and directory access service calls is made available to users as part of the operating system to use the secondary storage. A second abstraction presented by mass storage is as part of the virtual memory system, where "pages" of block data with virtual addresses may be kept on disk and swapped in and out of main memory as needed. When a page request is made for data that is not found in memory, a page fault is indicated and the operating system performs the necessary tasks to make room for the requested page in main memory by moving a less-used page on to disk and then bringing the desired page into memory while updating various tables. This is performed transparently to the user, but can take more than a million times longer than a similar data access request to cache. Some SMP nodes, especially those used as subsystems of commodity clusters or MPPs, may not include their own secondary storage. Referred to as "diskless nodes", these will instead share secondary storage which is itself a subsystem of the supercomputer or even external file systems shared by multiple computers and workstations. Diskless nodes are smaller, cheaper, lower energy, and more reliable.

Every SMP has multiple I/O channels that communicate with external devices (outside the SMP), user interfaces, data storage, system area networks, local area networks, and wide area networks, among others. Every user is familiar with many of these, as they are also found on deskside and laptop systems. For local area and system area networks, interfaces are most frequently provided to Ethernet and InfiniBand (IB) to connect to other SMPs of a larger cluster or institutional environments such as shared mass storage, printers, and the internet. The universal serial bus (USB) has become so widely employed for diverse purposes, including portable flash drives, that it is ubiquitous and available on essentially everything larger than a screen pad or laptop, and certainly on any deskside or rack-mounted SMP. JTAG is widely employed for system administration and maintenance. The Serial Advanced Technology Attachment (SATA) is widely used for external disk drives. Video graphics array and high-definition multimedia interface provide direct connection to high-resolution video screens. There is usually a connection specifically provided for a directly connected user keyboard. Depending on the system, there may be a number of other I/O interfaces.

6.3 AMDAHL'S LAW PLUS

Consider a situation analogous to one that dominates computing: whether to fly or drive to get from one city to another nearby city. The airplane travels about 10 times faster than a car. At first thought, it would be obvious that flying is better than driving with a peak performance gain of an order of magnitude. But door to door may not be an advantage: it takes about the same amount of time either way. The overheads of getting to and from the airport, the waiting time at the airport, waiting at baggage claim, the delay in getting a rental car or taxi, and even the time to check in at the hotel all degrade the positive effect of having a significant accelerator (the jet airplane versus the automobile) over the majority of the distance. A very similar situation dominates computing: it is codified in an observation made by Gene Amdahl, and is appropriately referred to as "Amdahl's law".

As previously presented, the SLOW performance model identifies key factors that determine delivered (or sustained) performance, including parallelism (starvation), latency, overheads, and contention (waiting for arbitration for shared resources). The effective operation of SMP-class architecture can be measured as the ratio of the delivered performance to the theoretical peak performance of the system. Amdahl's law is an important relation that captures a critical aspect of the SLOW performance model, specifically the effect of the program parallelism that provides the performance gain. If, with some simplification, a computation is divided between the part or fraction (f, where $0 < f < 1$) that can benefit from acceleration, such as the number of processor cores available for parallel execution and the remaining part of the computation $(1 - f)$ that is forced to perform at the rate of a single-thread execution, a total performance gain, S, with respect to the full computation being performed at sequential speed can be determined. This is illustrated in Fig. 6.2.

In Fig. 6.2 the upper line represents the computation being performed in a purely sequential manner from start to end over a period of T_0. The part of the total execution that is available for acceleration, shown as the lighter shaded line, is T_F where $T_F < T_0$. The fraction of the computation that can benefit from performance gain is $f = T_F/T_0$. With acceleration applied to the fraction designated, the total speedup is $S = T_0/T_A$, where T_A is the time to solution of the accelerated code. The derivation for S is shown in Eqs. (6.2)–(6.6), where g is the gain of the accelerator over the conventional execution rate. This is known as Amdahl's law:

$$S = T_0/T_A \tag{6.2}$$

$$f = T_F/T_0 \tag{6.3}$$

$$T_A = (1 - f) * T_0 + \frac{f}{g} * T_0 \tag{6.4}$$

$$S = \frac{T_0}{(1 - f) * T_0 + \frac{f}{g} * T_0} \tag{6.5}$$

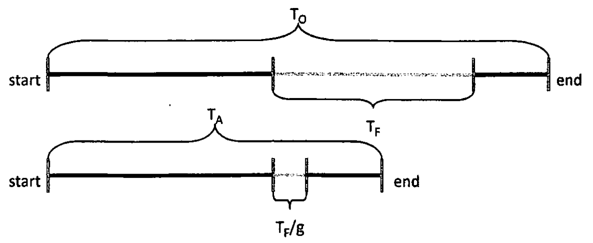


FIGURE 6.2

The time required to complete the serial execution (T_0) and accelerated (parallel) execution (T_A) of an application. A fraction of the application can be accelerated, indicated in green (light gray in print versions), requiring a non-accelerated time of T_F and an accelerated time of T_F/g. The total performance gain of the acceleration, S, is T_0/T_F.

$$S = \frac{1}{1 - f + \frac{f}{g}} \tag{6.6}$$

where T_0 is time for the nonaccelerated computation; T_A is time for the accelerated computation; T_F is time of the portion of computation that can be accelerated; g is peak performance gain for the accelerated portion of computation; f is fraction of the nonaccelerated computation to be accelerated; and S is speedup of the computation with acceleration applied.

The fundamental consequence of Amdahl's law is that independent of the size of the accelerator's peak performance gain, g, the sustained performance is bounded by the fraction, f, of the original code that can be accelerated. As a trivial limit, imagine that you have an accelerator capable of instantaneous execution no matter what the code, and that half the problem can be executed this way; that is, consider the case of infinite gain. The speedup for infinite gain and $f = 0.5$ is only $S = 2.0$. Fig. 6.3 shows the speedup with respect to the fraction of code accelerated for several values of g.

It is clear from the set of curves in Fig. 6.3 that sustained speedup is highly sensitive to the fraction of the computation that can benefit from acceleration. Where the fraction, f, to which g can be applied is less than 0.5 or so, S remains relatively low even if g is greater than an order of magnitude. It is only when f approaches 1.0 that dramatic reductions of time to solution result. For an SMP comprising p processor cores, g can be approximated by p_A, which is the number of cores applied to the parallel segments of the code ($p_A \leq p$).

Example

What is the minimum number of processor cores one must employ in an SMP to achieve a speedup of 3× where 75% of the user application can be fully parallelized?

Here $S = 3$, and $f = 0.75$ where we are seeking p_A as the minimum value of g required. Using the formulation for Amdahl's law derived above, the calculation follows:

$3 = 1/(1-0.75 + (0.75/g))$ or $g = 9$.

At least nine cores of the SMP must be used to get a speed up of 3× with this code.

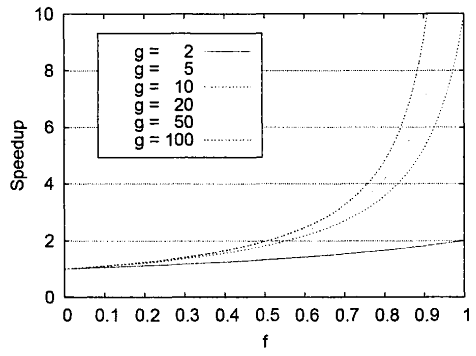


FIGURE 6.3

The speedup with respect to the fraction of code accelerated for several values of g.

But that is the good news. There are other sources of performance degradation that also come into play; in particular the overhead, v, of managing the parallel tasks, which does not contribute to the real work but does add to the critical time to solution. The timelines in Fig. 6.2 suggest that all the work that can be accelerated occurs in one large chunk, when in reality it is usually partitioned in a sequence of chunks, each controlled by some amount of overhead work, as shown in Fig. 6.4.

As seen in the top sequential (nonaccelerated) timeline of Fig. 6.4, the fraction of the computation that can be accelerated is broken into $n = 4$ partitions, which together make up the fraction f of the total work that can be accelerated. If this were the only difference, with a bit of manipulation the formulation of speedup would remain the same as that originally derived. However, for each partition of code to be accelerated, there is hopefully only a small amount of overhead work added to the critical path of execution time. Unfortunately, the size of the overhead is usually relatively constant independent of the granularity of the parallelized useful work parts. Further, the more partitions, n, into which the work is

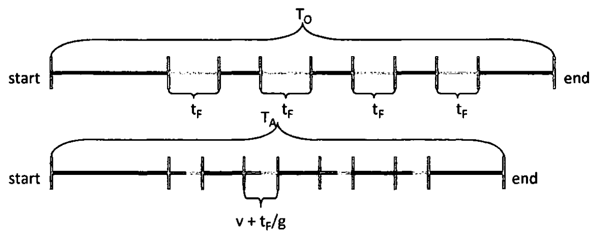


FIGURE 6.4

Timelines for nonaccelerated (T_O) and accelerated (T_A) executions, similar to Fig. 6.2 but including overhead, v.

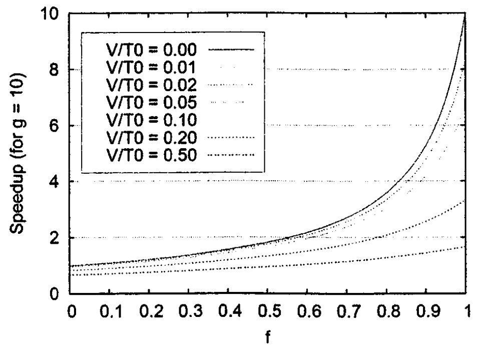


FIGURE 6.5

The speedup for various overhead ratios as a function of the fraction of code that can be accelerated for a fixed gain ($g = 10$).

divided, the more the additional overhead. By considering this overhead, a new extended version of Amdahl's law is derived in Eqs. (6.7)–(6.9).

$$T_A = (1-f) * T_0 + \frac{f}{g} * T_0 + n * v \tag{6.7}$$

$$S = \frac{T_0}{T_A} = \frac{T_0}{(1-f) * T_0 + \frac{f}{g} * T_0 + n * v} \tag{6.8}$$

$$S = \frac{1}{(1-f) + \frac{f}{g} + \frac{n * v}{T_0}} \tag{6.9}$$

where v is the overhead of an accelerated work segment and V is total overhead for the total accelerated work, $\sum_i^n v_i$.

Using the new equation for speedup given in Eq. (6.9), there is a new ratio added to the denominator that is proportional to the overhead v and the number of partitions, n. If there is no overhead ($v = 0$), the results are the same as the original formulation of Amdahl's law. If there is only a single large fraction of the code to be accelerated ($n = 1$), the result is almost the same. But as the parallelism is increased in the number of separate components, the overhead has an increasingly degrading effect. This is shown in Fig. 6.5.

As in Fig. 6.3, the abscissa axis in Fig. 6.5 is the fraction of code that can be accelerated, f, but g here is constant for all curves. A new independent variable, overhead v, is added to the plot, while T_0 is constant. As the overhead increases, performance gain, S, is reduced.

6.4 PROCESSOR CORE ARCHITECTURE

The modern multicore processor, sometimes called a "socket", consists of a number of cores, a potentially complex cache hierarchy, one or more interfaces to external main memory and I/O buses, and ancillary logic. While differing in details, most common processors can be characterized by a

Table 6.2 Characterization of Several SMP Processors

Processor	Clock Rate	Caches (per Core)	ILP (Each Core)	Cores Per Chip	Process and Die Size	Power (W)
AMD Opteron 6380	2.5 GHz (3.4 GHz turbo)	L1I: 32 KB L1D: 16 KB L2: 1 MB L3: 16 MB total	4 FPops/cycle 4 intops/cycle	16	32 nm, 316 mm^2	115
IBM Power8	3.126 GHz (3.625 GHz turbo)	L1I: 64 KB L1D: 32 KB L2: 512 KB L3: 8 MB L4: 64 MB total	16 FPops/cycle	12	22 nm, 650 mm^2	190/247
Intel Xeon E7-8894V4	2.4 GHz (3.4 GHz turbo)	L1I: 32 KB L1D: 32 KB L2: 256 KB L3: 60 MB total	16 FPops/cycle	24	14 nm	165

shared set of parameters, shown in Table 6.2. Among these are the number of cores per socket, the size and interconnectivity of the cache levels (usually two or three levels), the clock rate of the core, the number and type of arithmetic logic units per core (ILP), the die size (between one and four square centimeters), the feature size, and the delivered performance for one or more standardized benchmarks. For the application programmer many of the details may not matter, but the rate at which instructions are issued, the number of operations performed per instruction issue, the average time per memory access, and the delays due to I/O requests are principal in determining the delivered performance. In this section the major structures of the processor core are described and how they contribute to achieved performance. Section 6.5 examines the memory and cache hierarchy in depth to understand the role of locality in reducing average memory access time.

6.4.1 EXECUTION PIPELINE

The earliest generation of sequential computers issued and completed one instruction at a time using the oft-quoted "fetch—execute—writeback" cycle. With the low clock rates possible in early vacuum-tube and transistor technologies, this was satisfactory. But as clock rates improved with advanced technologies (e.g., small- and medium-scale integration), this straightforward approach became untenable. The complexity of the full issue to completion of instructions required too many layers of logic, with the resulting latency bounding the feasible clock rate.

A pipelined structure was adopted to partition the full compute operation into a sequence of microoperations which together achieved the same functionality. The time from instruction issue to completion would actually be longer than for a single logical function of the same purpose, but each stage of the pipeline would take much less time. As the clock rate was limited by the instruction issue cycle time, which was itself determined by the propagation delay through the longest stage of the pipeline, an execution pipeline with as many stages as possible each of the same delay allowed the clock rate to be increased appreciably. Early execution pipelines with four or five stages were eventually superseded by much longer pipelines.

As discussed in Chapter 2, pipeline logic structure is a general way of exploiting a form of very fine-grain parallelism, as each pipeline stage operates simultaneously. Ideally, the parallelism of a functional pipeline is equal to the number of stages of which the total pipeline is formed. Execution pipelines benefited from both the reduction in clock cycle times and the parallelism of their constituent stages. But a number of other factors imposed limits on the degree of pipelining that could be effectively employed. Among these are:

1. The size of the total function limited the number of logic layers that were required and thus the maximum number of stages into which the pipeline could be divided.
2. Imbalance in the number of logic layers in each stage made some stages slightly longer than others and therefore slowed down the rate at which signals could propagate through the execution pipeline.
3. The overhead of the interface between successive stages of the pipeline added additional propagation delay to each stage, bounding how fast the signals could proceed through the execution pipeline.
4. Not all execution functions were the same, and they did not necessarily require the same number of function stages. Those requiring more stages would waste some of the hardware when other execution cycles required fewer stages.
5. Intermediate values of one operation might be required for a following operation, but would be unavailable in time for the succeeding operation to be issued at its earliest opportunity. Alternatively, it would stall waiting for the results of the earlier operation to complete.
6. Conditional operations complicate the efficient use of an execution pipeline. Their function is to perform a branch to a noncontiguous instruction location, but to do so only if a predicate value is true, which must also be determined. This extends the number of microaction sequences, disrupting the flow within the execution pipeline and causing delays or "bubbles" to be inserted, thus slowing down execution.

To speed up execution despite these inhibiting factors, the core architecture has evolved in a number of forms and functions, briefly described in the following subsections.

6.4.2 INSTRUCTION-LEVEL PARALLELISM

Superscalar architectures enable multiple operations to be launched by a single instruction issue. This is achieved through the incorporation of multiple arithmetic logic units (ALUs), including both floating-point and integer/logical functional units, among others. Additional single-instruction multiple data units may be included to perform the same operations on multiple data values from the same instruction. Known as ILP, this provides among the finest-grain parallelism available to a processor core, and for special cases it can have a dramatic impact on total throughput. Unfortunately, experience over more than two decades shows that in general such peak capabilities are rarely exhibited while still adding complexity, overhead, and power demand to the advanced designs.

6.4.3 BRANCH PREDICTION

The problem with conditional branch instructions is discussed in Section 6.4.1. To eliminate bubbles caused by the delay between determining the Boolean value of the predicate and committing the virtual address of the next instruction to be executed, a statistical approach known as "branch prediction" is employed. As the name implies, upon a branch instruction being issued, the hardware makes a guess as to which of the two alternative instructions that may be followed will be issued. There is a long history of techniques to do this, and further reading on this topic is found in the bibliography [2–6]. But for this discussion the key idea is that depending on the role of the particular branch prediction, one of the two paths is more likely. For example, if a branch is used at the bottom of a loop, it is far more likely that the predicate will redirect the execution flow to the top of the loop rather than immediately continuing on. If a branch is associated with error handling, it is highly unlikely that this path will be pursued and more likely that the next instruction to be issued will be part of the regular computation stream. There will always be cases where the wrong choice is made, so the hardware architecture has to be capable of rolling back the computation to take the other path; this itself is a large body of architecture lore. Some codes, like system software, are very heavy with branches, and in such cases branch prediction architecture support can go a long way in improving efficiency.

6.4.4 FORWARDING

Key to the concept of the execution pipeline is that the time to issue successive instructions is potentially far shorter than the time to completion through the many stages of the pipeline. It is possible that two succeeding instructions may impose one or more precedence constraints, such that the second instruction requires as arguments the result value of the preceding (first) instruction issued. Usually an instruction will acquire its operands from the core's register set. But in the condition described there will not have been enough time for the resulting value of the first instruction to be calculated and written back into the register bank before the second instruction would ordinarily read the same value from the register in which this intermediate value resides. The solution is "forwarding". Forwarding means added data transfer channels that move data from downstream execution pipeline segments to the appropriate upstream segment, making the argument value available in time for the instructions to follow more closely in succession. Combined with compiler reordering where necessary gaps can be filled with one or more unrelated instructions, pipeline stages can be filled and bubbles eliminated through forwarding.

6.4.5 RESERVATION STATIONS

Different operations take different amounts of time to complete, and the execution pipeline becomes multipath with shorter links in a simple Boolean logic operation than for a floating-point multiply. If strict ordering were preserved, i.e., the order of completion was forced to be identical to the order of issue, the rate of instruction processing would be constrained by the slowest operations with repeated stalls of backstream instructions. This problem is addressed by reservation stations, a concept dating back to the late 1960s, and the ideas of data flow in the 1970s. A reservation station is a special-purpose buffer register, invisible to the user, which temporarily holds a previous result value. Its special feature is that it "knows" what follow-on instructions require the captured value, and those instructions know the corresponding reservation station(s) from which to acquire their argument values. If the instruction tries to get the operand value before it is available in the designated reservation station, the instruction will be delayed at the reservation station but will not impede the progress of the execution pipeline. There are many alternative architecture methods by which this complex out-of-order scheduling mechanism can be achieved (often referred to as the "Tomasulo algorithm"), but in every case the use of reservation stations permits substantial flexibility in operation of the execution pipeline and greater efficiency.

6.4.6 MULTITHREADING

So far the discussion of the processor core's execution pipeline assumes a single stream of instructions, each with one or more associated operations. While these can prove complex in detail, they still are based on the original von Neumann concept of a single program counter (or instruction pointer) that is incremented for each instruction issue except for branch instructions. This is a clean and elegant approach but suffers from a number of edge conditions, such as those previously discussed. Many of these problems are due to the interrelationships among adjacent or neighboring instructions of a single instruction stream. One way to address this challenge in a single processor core was introduced by Burton Smith in the 1980s: the concept of "multithreading". In its simplest version, multithreading incorporates multiple instruction streams or threads routed through sets of multiple instruction pointers and their associated register sets. The rest of the execution pipeline is shared, and a round-robin instruction issue scheduler selects each successive instruction fetch from different threads. This hides the latency of the execution pipeline and, if sufficient threads are employed, the latencies to main memory as well.

BURTON SMITH AND THE MTA

Photo by Dimitrij Krepis via Wikimedia Commons

Burton Smith is a leading computer architect and is considered the father of multithreading architecture, for which he was awarded the Eckert-Mauchly Award and the Seymour Cray Award. As a cofounder of Denelcor and later the founder of Tera Computer Company Smith led the commercialization of multithreaded architecture. In 2000 Tera became Cray, with the merger of the Cray Research business unit of Silicon Graphics. Burton Smith was the chief architect of the Tera MTA (multithreaded architecture), a breakthrough design that continues to inform high performance computer development. Burton Smith became a technical fellow of Microsoft in 2005, where he remains to this day advancing future technologies and computing concepts.

The MTA-1 was deployed at the San Diego Supercomputer Center; the initial system was unique in that it was implemented using very high-speed logic based on gallium arsenide. This architecture incorporated four processors each with 128 independent register sets and program counters, permitting a total of 512 threads to be executed simultaneously. Each processor integrated a high-speed arithmetic processing unit to which its local threads could apply operations to be performed, thus sharing the ALU for maximum utilization, efficiency, and performance. The MTA's strength was in its ability to hide memory access latencies from the arithmetic units and adjust to asynchronies of operation. This eliminated the needs for data caches, precluding the complexities and costs of achieving consistency among them. Empty/full bits on every word and a use of tagged memory enabled fine-grain synchronization. Other tags made possible control semantics such as futures, among others. The MTA-1 prototype was followed by a much less expensive and more densely packed CMOS version, the MTA-2, with further advances leading to the Cray XMT System in 2009.

6.5 MEMORY HIERARCHY

The "memory wall", alternatively termed the "von Neumann bottleneck", recognizes the mismatch between the peak demand rate of the processor socket for data access and the possible delivered throughput and latency of the main memory technology, principally semiconductor DRAM. As demonstrated in Fig. 6.6, performance gain for processors increased on average by 60%/year, while that of main memory experienced only about a 9%/year improvement. Over time this has led to a two order of magnitude difference between processor speeds and memory speeds. To address this challenge, and indeed move even further into the domain of secondary storage and beyond, computer architecture in general and SMP architecture in particular have evolved a hierarchical structure of a sequence of layers of storage components with increasing density and capacity in one direction of the hierarchy, and greater access speed including higher bandwidth and lower latency in the other direction.

6.5.1 DATA REUSE AND LOCALITY

Fundamental to the success of this memory architecture is the strategy of data reuse through locality. If a value of a variable is used by a program repeatedly and frequently, storing it in a very high-speed memory device very close to the processor core will deliver near-peak performance. This is "temporal locality", which reflects the property of data that associates the probability of usage with recent prior usage. High temporal locality suggests that a particular variable is accessed frequently in a moderate period of time. Low temporal locality indicates that a variable is probably only used once or a couple of times in the moderate contiguous period, if accessed at all. A second form of locality often exploited is "spatial", which indicates an association of locality among adjacent or near neighbors in contiguous address space. High spatial locality suggests that the probability of a variable (virtually addressed value) being accessed is higher if one of its adjacent or neighboring variables has been recently accessed. These two forms of locality concerning the reuse patterns of virtually addressed

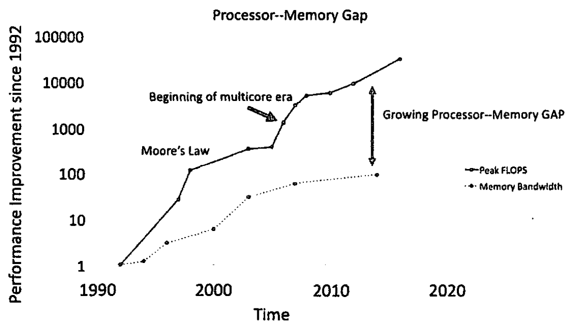


FIGURE 6.6

Performance gains for processors increased by four orders of magnitude while main memory experienced an improvement of only two orders of magnitude during the same period of time.

variables provide the foundations for the structure and operation of the memory hierarchy to mitigate the effects of the discrepancies between bandwidths and latencies of processor and memory technologies.

The second factor of practical concern is the tradeoff relationships between the characteristics of storage capacity per unit area, cycle time of access, and power consumption. In Chapter 17, it is shown that diverse on enabling technologies it is shown that diverse data storage technologies vary in terms of these parameters. In general, faster memory technologies take up more room on a semiconductor die or other medium for the same amount of storage while consuming greater power. It is impractical to create a main memory layer that is big enough to hold all the software and data required for a given user application while running fast enough to keep the processor cores fully utilized at their peak instruction issue throughput.

6.5.2 MEMORY HIERARCHY

The conventional way for modern computing architectures, including SMP systems, to address these tradeoffs through exploitation of data locality is in the structure of the memory hierarchy, also known as the memory stack.

As shown in Fig. 6.7, the memory hierarchy or stack consists of layers of memory storage technology, each with different tradeoffs between memory capacity, costs, and cycle times, which reflect bandwidths and latencies. By far the slowest but also the highest capacity is the use of tape archival storage, often consisting of possibly thousands of tape modules physically stored in a robotic library with total capacities approaching exabytes. But in an unloaded system access to stored data could take upwards of a minute, even though the cost of a megabyte is a fraction of a cent. Tape robots provide part of mass storage called "tertiary storage"; another part of mass storage is secondary storage made up of hard-disk drives (HDDs). Disks, like tapes, use a magnetic storage medium. But unlike tapes, which present one long serial stream of storage that can take a long time to go from one end to the other, data on disks are laid out in concentric rings (called "cylinders") that spin on an axis. A radial arm moving in and out across the spinning disk selects the appropriate cylinder and waits for the required data to come around to be detected by the arm's head. A typical disk drive may hold several terabytes, deliver data at a peak streaming rate of 300 MB/s, and impose an overall access time of around 10 ms. While this is 100,000 times longer than access to main memory, it may hold 1000 times as much stored data and exhibit a latency 10,000 times shorter than tape drives. A third technology recently introduced commercially, nonvolatile random access memory, is increasingly being employed as a partial replacement for disk drives for much faster response than disks but slower than main memory. Mass storage is usually presented to the user in the form of logical data modules called files, and directories that hold files as well as other directories.

At the other (top) end of the memory hierarchy are the processor core registers that operate at the speed of the clock rate and support multiple access ports allowing multiple reads and writes into/out of the register banks in each instruction cycle. While operating at native processor speeds, registers take up a lot of room and consume significant energy per cycle. Registers also exhibit their own address space not associated with the memory address namespace. The instruction-set architecture (ISA) is logically structured such that data explicitly moves between identified registers and the variables in the main memory.

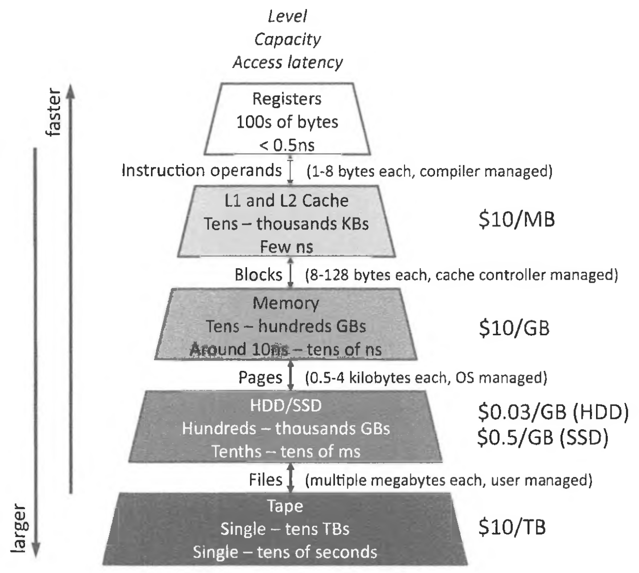


FIGURE 6.7

The memory hierarchy delineated by memory capacity, cost, and cycle times.

Courtesy David A. Patterson

The main memory is provided by DRAM semiconductor devices. Many such components are mounted on personal computer cards plugged into sockets compliant with industry-standard interfaces. As much as 4 gigabytes of memory per processor core for an SMP is often provided, although this can drop to as low as 1 gigabyte per core depending on the number of cores per socket. But access times from register to DRAM can be between 100 and 200 clock cycles, far too much for effective computing.

Between processor core registers and SMP main memory modules is a cache system to impedance match between these two extremes in timing and bandwidth. Logically the cache system is transparent to the user, in that it is not separately addressable but instead accepts memory access requests. If the variable address requested by a core has a copy of the variable value somewhere within its cache, the cache provides that value in the case of a load operation to the designated core register. If a copy does not exist, the cache system automatically passes the request to the main memory to perform the data access. Where data locality applies, a cache "hit" is likely and the access time will be that of high-speed cache rather than the slower main memory, which can be as much as two orders of magnitude faster.

In a modern SMP the cache is usually not a single layer of higher-speed memory, but rather multiple layers to find an optimal balance of speed and size. Typically there are three layers: L1, L2, and L3. L1 is the fastest and smallest, and usually consists of two separate caches: one for data and the other for instructions to provide enough peak bandwidth. L2 is slower but much larger, and like L1 is made from SRAM circuits. The L3 cache is much larger than the lower-layer caches, but is slower.

Unlike the first two, L3 is usually a separate chip of DRAM rather than SRAM circuits to achieve the greatest density.

A simple hierarchical structure would provide each core with its own separate L1, L2, and L3 caches. However, often multiple cores are working on the same set of data, and the maximum amount of data in a layer could be increased by allowing more than one core to share at least some of the cache. Typically the L1 cache is not shared due to the need for maximum individual bandwidth. Also typically the L3 cache is shared among the processor cores or some subset of them. L2 caches may be either dedicated to a single core or shared among two or more cores. Part of the tradeoff is about bandwidth and possible contention for cache access among sharing cores.

6.5.3 MEMORY SYSTEM PERFORMANCE

It is clear that the time to access a value from a specified variable in the memory system will vary dramatically depending on a number of factors, most specifically where the closest copy of the value is in the memory hierarchy. While analyzing such a complex memory architecture can be very complicated due to the number of levels, the overheads involved, issues of contention, and so forth, a simplified version of the problem still exposes the principal tradeoffs and shows how dramatically the average memory access time can change depending on the hit rates to cache as a consequence of locality. For this purpose, the cache is assumed to be a single intermediate layer between processor core registers and main memory. Without a detailed queuing analysis or similar in-depth model, operational metrics are adopted to capture the specific properties of the architecture and application memory access profile, plus a quality metric of performance. An analytical model is derived to show the sensitivity between delivered performance and the effectiveness of caching.

The quality metric of choice in this case is *CPI* or cycles per instruction. Time to solution, *T*, is proportional to cycle time, T_{cycle}, and the number of instructions to be executed for a user task, I_{count}. Because the purpose of this analysis is to expose the implications of memory behavior, the instruction count is partitioned between those instructions associated with the number of register-to-register *ALU* instructions, I_{ALU}, and the number of memory access instructions, I_{MEM}. For each of these two classes of instructions there is a separate measure of cycles per instruction, one for the register-to-register *ALU* operation, CPI_{ALU}, and one for the memory instructions, CPI_{MEM}. The total value for time, I_{count}, and *CPI* can be derived from the breakdown between *ALU* and memory operations according to Eqs. (6.10)–(6.12).

$$T = I_{count} * CPI * T_{cycle} \tag{6.10}$$

$$I_{count} = I_{ALU} + I_{MEM} \tag{6.11}$$

$$CPI = \frac{I_{ALU}}{I_{count}} * CPI_{ALU} + \frac{I_{MEM}}{I_{count}} * CPI_{MEM} \tag{6.12}$$

The full set of parameters is defined as:

T, total execution time; T_{cycle}, time for a single processor cycle; I_{count}, total number of instructions; I_{ALU}, number of ALU instructions (e.g., register to register); I_{MEM}, number of memory access instructions (e.g., load, store); *CPI*, average cycles per instruction; CPI_{ALU}, average cycles per ALU instruction; CPI_{MEM}, average cycles per memory instruction; r_{miss}, cache miss rate; r_{hit}, cache hit rate; $CPI_{MEM-MISS}$, cycles per cache miss; $CPI_{MEM-HIT}$, cycles per cache hit; M_{ALU}, instruction mix for ALU instructions; and M_{MEM}, instruction mix for memory access instructions.

The idea of an instruction mix simplifies representation of this distinction between ALU and memory operations, providing ratios of each with respect to the total instruction count.

In addition, the parameter that expresses the effect of data reuse is defined as the hit rate, r_{hit}, which establishes the percentage of time that a memory request is found in the cache. The opposite of this parameter can be useful: $r_{miss} = (1 - r_{hit})$. One last distinction is made for CPI_{MEM} depending on whether a hit or a miss occurred. These represent the costs, measured in number of cycles of memory instruction access times, depending on whether there was a hit or a miss at the cache. $CPI_{MEM\text{-}HIT}$ is a fixed value of the number of cycles required for an access that is served by the cache, and $CPI_{MEM\text{-}MISS}$ is the cost in cycles of going all the way to main memory to get a memory request serviced in the case of a cache miss. The relationships among these distinguishing parameters are demonstrated in Eqs. (6.13)–(6.17), associating them with the definition of full execution time.

Instruction mix:

$$M_{ALU} = \frac{I_{ALU}}{I_{count}} \tag{6.13}$$

$$M_{MEM} = \frac{I_{MEM}}{I_{count}} \tag{6.14}$$

$$M_{ALU} + M_{MEM} = 1 \tag{6.15}$$

Time to solution:

$$CPI = (M_{ALU} * CPI_{ALU}) + (M_{MEM} * CPI_{MEM}) \tag{6.16}$$

$$T = I_{count} * [\,(M_{ALU} * CPI_{ALU}) + (M_{MEM} * CPI_{MEM})] * T_{cycle} \tag{6.17}$$

Finally, the values for CPI_{MEM} and T as functions of r_{miss} are presented in Eqs. (6.18) and (6.19). It may appear peculiar that the coefficient of $CPI_{MEM\text{-}HIT}$ is not r_{hit}. This is because the cost of getting data from or to the cache occurs whether or not a miss occurs.

$$CPI_{MEM} = CPI_{MEM-HIT} + r_{miss} * CPI_{MEM-MISS} \tag{6.18}$$

$$T = I_{count} * [(M_{ALU} * CPI_{ALU}) + M_{MEM} * (CPI_{MEM-HIT} + r_{miss} * CPI_{MEM-MISS})] * T_{cycle} \tag{6.19}$$

This shows the effect of the application-driven properties, including I_{count}, M_{MEM}, and r_{miss}. Architecture-driven properties are reflected as T_{cycle}, $CPI_{MEM\text{-}MISS}$, and $CPI_{MEM\text{-}HIT}$ in determining the final time to solution, T.

Example

As a case study, a system and computation are described in terms of the set of parameters presented above. Typical values are assigned to these to represent conventional practices, architectures, and applications. These are shown below.

I_{count}	$=1E11$
I_{MEM}	$=2E10$
CPI_{ALU}	$=1$
T_{cycle}	=0.5 ns
$CPI_{MEM-MISS}$	$=100$
$CPI_{MEM-HIT}$	$=1$

The intermediate values for instruction mix are computed as follows:

$$I_{ALU} = I_{count} - I_{MEM} = 8E10$$

$$M_{ALU} = \frac{I_{ALU}}{I_{count}} = \frac{8E10}{1E11} = 0.8$$

$$M_{MEM} = \frac{I_{MEM}}{I_{count}} = \frac{2E10}{1E11} = 0.2$$

This example shows the impact of the cache hit rate on the total execution time, which can prove to be one of the most important determining factors of application time to solution and one of which the user has to be aware as data layout is considered. Two alternative computations are considered. The first is favorable to a cache hierarchy (this example simplifies, with only one layer) with a hit rate of 90%. With this value established, the time to solution can be determined as shown in Eqs. (6.20)–(6.22).

$$r_{hit\,A} = 0.9 \tag{6.20}$$

$$CPI_{MEM\,A} = CPI_{MEM-HIT} + r_{MISS\,A} * CPI_{MEM-MISS} = 1 + (1 - 0.9) * 100 = 11 \tag{6.21}$$

$$T_A = 1E11 * [(0.8 * 1) + (0.2 * 11)] * 5E10 = 150\text{ s} \tag{6.22}$$

But if the cache hit rate is lower, in this case 50%, a recalculation with this new value shows a dramatic reduction of performance, as shown in Eqs. (6.23)–(6.25).

$$r_{hit\,A} = 0.5 \tag{6.23}$$

$$CPI_{MEM\,B} = CPI_{MEM-HIT} + r_{MISS\,B} * CPI_{MEM-MISS} = 1 + (1 - 0.5) * 100 = 51 \tag{6.24}$$

$$T_B = 1E11 * [(0.8 * 1) + (0.2 * 51)] * 5E10 = 550\text{ s} \tag{6.25}$$

The difference is more than a factor of 3× performance degradation, just because of the change in the cache hit rate.

6.6 PCI BUS

The explosion of the personal computer market initiated in the late 1970s created a growing need for high performance industry-standard interfaces that would enable portability, reusability, and upgradeability of peripherals and custom expansion boards. While initially this void was filled by IBM's industry-standard architecture (ISA) bus, the bulkiness of its connector, low data bandwidth, poor expandability (limited number of interrupts and direct memory access channels), and cumbersome configurability forced the manufacturers to look for improved solutions. IBM's follow-up to ISA, the Micro Channel Architecture, did not gain widespread popularity due to being a proprietary standard that required licensing fees. Broad industry response resulted in the development of stopgap implementations such as extended ISA and Video Electronics Standards Association local bus, the latter of which was primarily used to satisfy the bandwidth requirements of newer graphics cards. The true breakthrough was the work of Intel's Architecture Development Laboratory that resulted in the specification of Peripheral Component Interconnect (PCI) in 1992 [7]. The PCI bus in somewhat modified form may still be found on some motherboards manufactured today.

The PCI bus operates independently from a processor's native memory bus and requires a bridge circuit to provide memory-mapped access from the CPU, as shown in Fig. 6.8. More than one bus may be supported via additional bridges. PCI originated as a parallel multidrop 32-bit bus in which multiple devices connect electrically to the single instance of control and data lines. Its signaling, timing, transaction protocol, mechanical connector properties, and power management have been defined and extended in a series of specifications, the last of which was version 3.0 released in 2004. The clocking frequency was originally set at 33 MHz for a peak bandwidth of 132 MB/s. Subsequent standard releases permitted a faster 66 MHz clock and added a 64-bit data bus option for peak data bandwidth of 533 MB/s. The signal levels were also reduced from 5 V to 3.3 V to reflect the prevailing trends in chip I/O standards and lower the required bus driver power levels. Extended PCI, later introduced to optimize certain aspects of PCI functionality in servers, increased the clock to 266 and 533 MHz, resulting in a maximum transfer rate of 4266 MB/s. To prevent potential card damage by using an incompatible implementation, these options were tied to differently keyed connectors (Fig. 6.9). A short-lived variant of PCI with a dedicated CPU-to-GPU (graphics processing unit) bus was called the Accelerated Graphics Port and featured its own connector type, incompatible with PCI.

To address several shortcomings of the conventional PCI bus architecture (poor scaling in a number of devices, interrupt sharing, an explosion in the number of pins, poor power characteristics of single-ended I/O, limited bandwidth scalability, and access synchronization issues), the PCI Special Interest Group, including among others Intel, Hewlett—Packard, Dell, and IBM, adopted in 2002 a new design called PCI Express (PCIe), with the system schematic shown in Fig. 6.10. Electrically, PCIe is a descendant of Intel's 3GIO (third-generation I/O) initiative that utilized multiple serial links operating

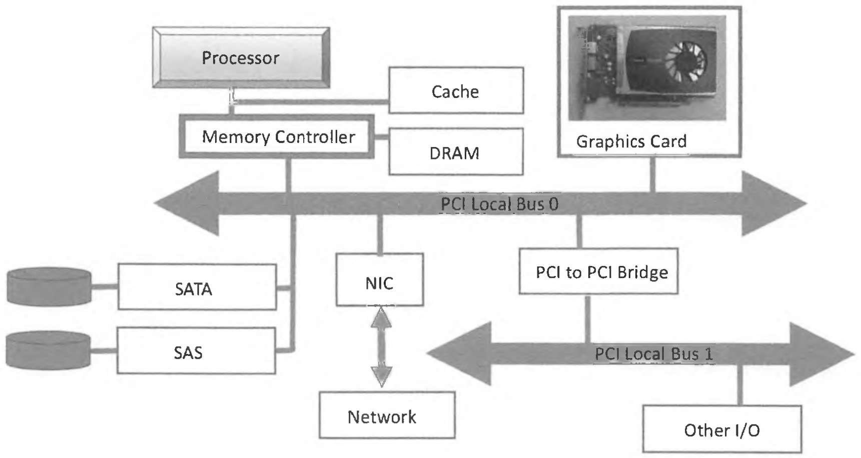


FIGURE 6.8

Layout of a system equipped with multiple PCI buses.

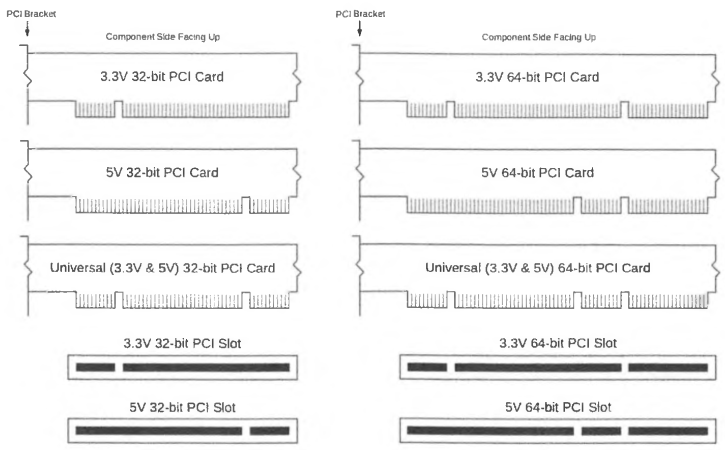


FIGURE 6.9

Connectors supporting different PCI variants.

Image via Wikimedia Commons

at 2.5 Gbps each. The links use low-voltage differential signaling, giving them very good noise immunity and reducing their electromagnetic interference (EMI) levels compared to single-ended operation. High bandwidth of a single link brings a much-needed reduction in the number of required pins: full-duplex connection requires only a pair of wires for transmit and another pair for the

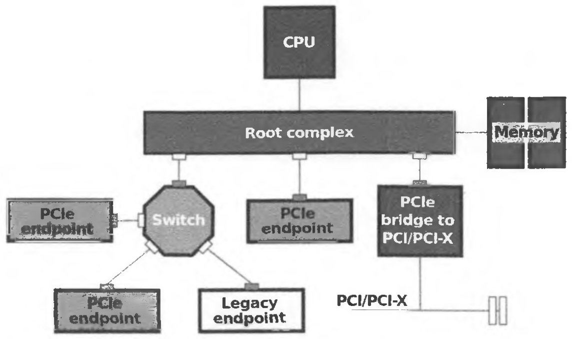


FIGURE 6.10

Diagram of a PCIe-equipped system.

By Miiu92 via Wikimedia Commons

receive function. Scaling to the desired bandwidth is accomplished by adding more links, called "lanes" in PCIe vernacular. Specifications permit up to 32 lanes per card slot, although practical implementations rarely exceed 16 lanes.

Three major revisions of PCIe specifications were released. The first defined operation at nominal 2.5 Gbps with 8 b/10 b encoding (each 8 bits of input data is converted into a 10-bit symbol on wire) for an effective peak of 250 MB/s per link. PCIe 2.0 increased the signaling rate to 5 Gbps per link, doubling the peak data bandwidth. Version 3.0 improved the encoding efficiency by using a 128 b/130 b scheme and further increased the wire rate to 8 Gbps, thus yielding 984.6 MB/s peak per lane, or 15.754 GB/s in aggregate when using a 16-lane device. Unlike conventional PCI, PCIe slots of different generations are backwards compatible, thus enabling the use of older cards in newer machines. PCIe permits plugging cards with fewer lanes into connectors providing more lanes; the inverse is also true as long as the connector can physically accept a card. It is also capable of sustaining the operation (at reduced bandwidth) even when some of the physical links fail. Fig. 6.11 compares sizes of PCIe connectors in various configurations.

PCIe connectors provide 12 V and 3.3 V supply voltages that may be used to power the connected cards. Slot-powered operation limits the power draw to only 25 W per board, which is insufficient to sustain the functioning of more demanding devices, such as GPUs. Such devices instead incorporate dedicated six-pin and eight-pin power connectors that plug into power supply harnesses to provide additional 75 W and 150 W circuits, respectively.

In terms of the protocol, PCIe inherits many properties of the original PCI. The communication is packet based. Each packet is either a posted request (writes data to target space), a nonposted request (initiates a read from the target), completion (carries the data read from the target space), or a

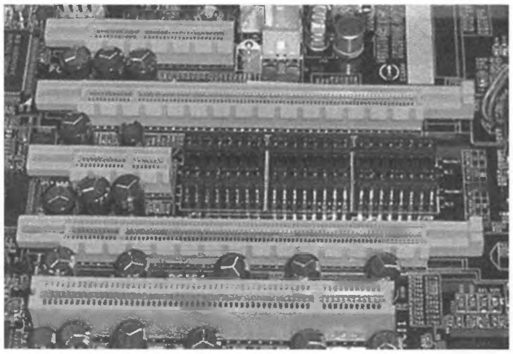


FIGURE 6.11

Comparison of PCIe slots with different lane counts (from the top: ×4, ×16, ×1, ×16) with a conventional PCI connector (bottom).

Via Wikimedia Commons

message that signals a specific event or supports a vendor-defined function. The elementary unit of transfer is a *double word* of 32bits. The transaction layer uses three or four double-word-long headers followed by the payload, which may contain up to 1024 double words (4 KB) of data. Larger data transfers must be split into multiple packets. The data-link layer wraps the transaction data with the packet sequence number and cyclic redundancy code sum used by the receiver to verify the packet integrity. The packets act on memory and I/O spaces (each can be independently configured to use 32- or 64-bit addresses) or the dedicated configuration space. PCIe devices may define up to six distinct read—write memory or I/O regions (fewer if 64-bit addressing is used) with different aperture (active address range) sizes, and a separate optional expansion read-only memory (ROM) space. The latter is used to provide device-specific information or, on compatible Intel platforms, to store additional boot code.

Today, PCIe is a dominant standard for attaching and high performance communication with expansion boards on different machines that may use a processor architecture other than the original Intel ×86 variant. The PCIe specifications are continuously updated and refined to reflect modern technological trends. The next revision of the standard, version 4.0, is expected to be finalized in 2017.

6.7 EXTERNAL I/O INTERFACES

The key I/O interfaces of an SMP are the network interface controllers, including Ethernet and IB, SATA for mass storage devices, JTAG for low-level hardware interface, and USB for connecting peripheral devices like keyboards. This section explores each in detail.

6.7.1 NETWORK INTERFACE CONTROLLERS

The two most common network interface controllers appearing in clusters in the Top 500 list of June 2016 were Ethernet and IB. The following subsections give a brief overview of these network interface controllers.

6.7.1.1 Ethernet

Named after a supposed medium for light propagation that was incorrectly thought to exist by many 19th century scientists, Ethernet is a standardized computer networking technology originally developed at Xerox's Palo Alto Research Center in 1973 by Robert Metcalfe, David Boggs, Chuck Thacker, and Butler Lampson [8]; it has since become ubiquitous. The Institute of Electrical and Electronics Engineers (IEEE) produced the official Ethernet standard 802.3 in 1983 and the technology continues to develop, reaching bandwidths of 100 Gbps.

Ethernet operates by breaking a stream of data into frames, with a preamble and start frame delimiter and ending with a frame check sequence. In the standard IEEE 802.3 Ethernet specification, the minimum frame size was 64 bytes and the maximum was 1518 bytes (since expanded to 1522 bytes). The preamble consists of 7 bytes followed by a single byte as a start frame delineator. The frame itself has a header containing the destination and source encoded in 48-bit addresses known as media access control (MAC) addresses. The frame data follows this header and is terminated by the frame check sequence. On Gigabit Ethernet networks jumbo frames of up to 8960 bytes can be used which bypass the standard Ethernet maximum of 1522 bytes.

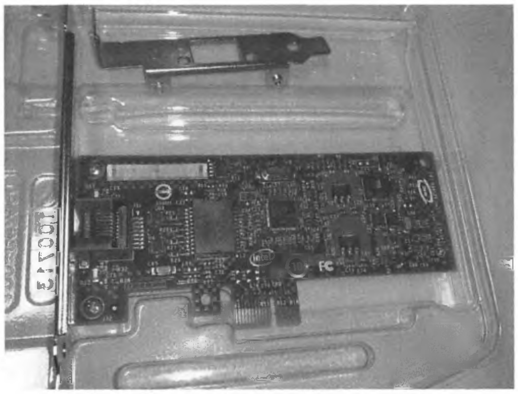


FIGURE 6.12

A Gigabit Ethernet network interface card.

By Dsimic via Wikimedia Commons

The state of the art for Ethernet is currently 100 Gbps. In the June 2017 Top 500 list of supercomputers, Gigabit Ethernet is featured in 207 systems and is the most common internal system interconnect technology in the list [9]. Examples of Gigabit Ethernet cards and switches are shown in Figs. 6.12 and 6.13.

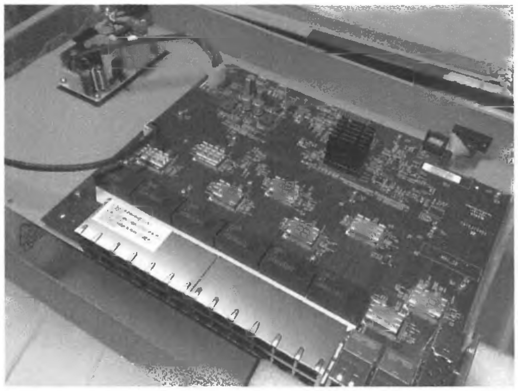


FIGURE 6.13

The internals of a Gigabit Ethernet switch.

By Dsimic via Wikimedia Commons

6.7.1.2 InfiniBand

IB is an alternative to Ethernet for computer networking technology and originated in 1999. Unlike Ethernet, IB does not need to run networking protocols on the CPU; these are handled directly on the IB adapters. IB also supports remote direct memory access between nodes of a supercomputer without requiring a system call, thereby reducing overhead. IB hardware is produced by Mellanox and Intel, with IB software developed through the OpenFabrics Open Source Alliance [10].

The state of the art for IB transfer rates is the same as the fastest transfer rate supported by the PCIe bus (25 Gbps for enhanced data rate). In the June 2017 Top 500 list of supercomputers, IB technology is the second most-used internal system interconnect technology, appearing in 178 systems [9]. Examples of IB cards and a port are shown in Figs. 6.14 and 6.15.

6.7.2 SERIAL ADVANCED TECHNOLOGY ATTACHMENT

SATA is a computer interface and communication protocol introduced in 2003. Its specifications are currently developed by the independent, nonprofit Serial ATA International Organization led by multiple industry partners, including dominant computing systems and storage manufacturers. It is used primarily to provide connectivity to mass-storage devices. SATA replaces the older parallel ATA (PATA) technology that was characterized by lower data transfer bandwidths, bulky ribbon cables frequently obstructing air flow in the node's case, and lack of proper support for hot-swapping of I/O

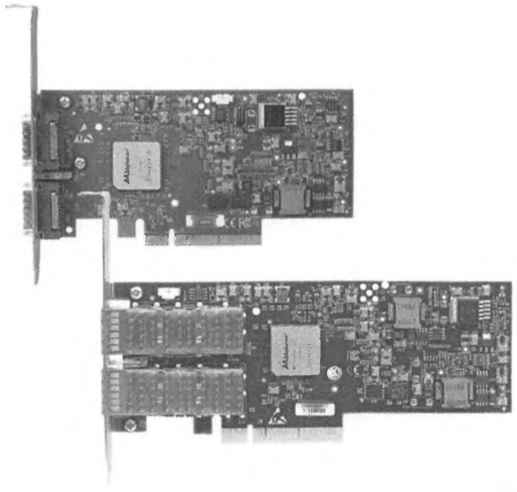


FIGURE 6.14

Mellanox IB cards.

Image courtesy Mellanox Technologies

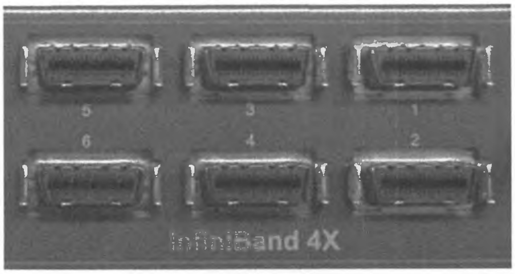


FIGURE 6.15

InfiniBand port.

By おむこさん志望 via Wikimedia Commons

devices. SATA interfaces may be found on most modern internal (i.e., housed inside the computer enclosure and therefore nonportable) HDDs, SSDs, and optical drives (CD-ROM, DVD-ROM, BD-ROM and their data-writer equivalents).

SATA supports only point-to-point topology between storage devices and controllers or port multipliers. SATA data connectors, shown in Fig. 6.16, contain only 7 pins compared to 40 mandated by PATA: one pair of wires for data transmission, a second pair of wires for data reception, and three ground connections. Data transmission is performed over high-speed serial links that use similar technology to PCIe and share many of the same quality characteristics with it. Serial links also take advantage of matched impedance cables, guaranteeing signal integrity over distances of at least 1 m. The power connectors utilize a 15-pin arrangement that provides ground reference and the 3.3, 5, and 12 V supply voltages needed by most of the attached devices to operate, and may also control

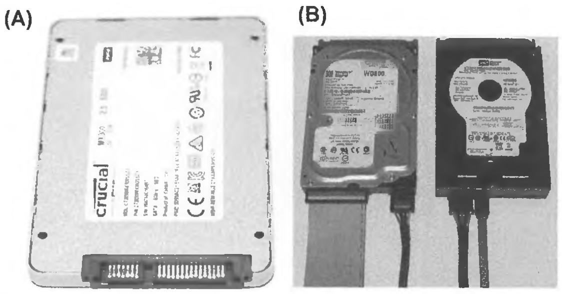


FIGURE 6.16

SATA connectors: (A) data (left, shorter) and power (right, longer) headers located on a 2.5" solid-state drive, and (B) older Parallel ATA cabling (left) contrasted with SATA (right).

staggered spin-up functionality. The latter is particularly useful in storage nodes populated with potentially dozens of disk drives, as enabling all of them at once would put a considerable strain on power supply during the power-up cycle, possibly reducing its useful lifetime. Both types of connectors use a two-phase mating sequence to ensure that the ground connection is made first and eliminate the possibility of unpredictable floating potentials during drive removal or insertion when the system is powered up. Most of the computer motherboards manufactured today support multiple SATA data ports (typically two to eight), while common power supplies provide multiple SATA-compatible hookups.

The first revision of SATA specifications supported a 1.5 Gbps signaling rate, resulting in a maximum peak data transfer rate of 150 MB/s. With the increases in HDD media speeds and the introduction of solid-state storage, this proved to be a serious performance bottleneck, and the next revisions, SATA 2.0 and 3.0, increased the raw signal rate to 3 and 6 Gbps, respectively. Modern chipsets are capable of detecting device speeds through autonegotiation and are backwards compatible with older drives. Early SATA 2.0 implementations, however, may require that the device is explicitly configured to the correct interface speed by setting a jumper on configuration pins and in some cases also by forcing proper basic input/output system settings. The newer SATA revisions also support native command queueing (NCQ), which may drastically improve the performance of I/O-intensive multitasking workloads by reordering the requests at physical block level, resulting in an overall shorter travel distance for the disk head. Other extensions included introduction of isochronous quality of service for periodically scheduled data accesses, host-side support for NCQ processing, and better power management. The specifications have been twice revised since (version 3.1 in 2011 and 3.2 in 2013), and defined additional interfaces, capabilities, and power management functions:

- mSATA interface for mobile devices
- M.2 small form factor standard
- microSSD standard for connectorless single-chip embedded storage
- "zero-power" state for idling optical drives
- TRIM command for SSDs that optimizes allocation of no longer used blocks on the device
- universal storage module for cable-free docking of portable storage modules
- required link power management, DevSleep, and transitional energy reporting for additional power savings
- rebuild assist that speeds up data reconstruction in redundant arrays of independent disks
- performance optimizations for solid-state hybrid drives
- signaling speed increase to 16 Gbps with a corresponding peak data rate of nearly 2 GB/s.

Besides the originally defined SATA data ports for internal I/O devices, several other form factors specified by the standard are already in widespread use or gaining popularity. The external SATA (eSATA) connector shown in Fig. 6.17 has been developed to provide connectivity to external storage devices. It features more robust connector and permits longer cables (up to 2 m) thanks to changes in required signal voltage levels. It is also shielded to reduce EMI emissions. eSATAp, or *powered* SATA, attempts to solve one of main shortcoming of eSATA, namely the necessity to provide a separate power source (and therefore an additional cable) to the external device. While not fully standardized yet, it aims to provide 5 and 12 V supply voltages as well as SATA and USB 2.0 data lines.

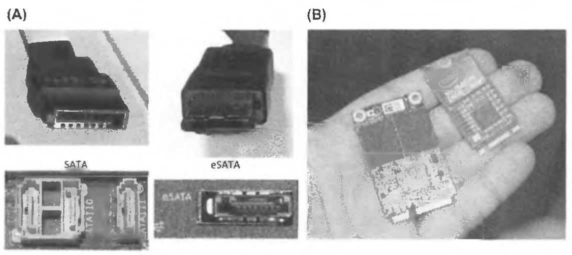


FIGURE 6.17

SATA interface variants: (A) eSATA compared to SATA; (B) mSATA (left) and M.2 (right) devices.

(B) Photo by Anand Lal Shimpi via Wikimedia Commons

Mini-SATA (mSATA) and its next revision, M.2 interfaces (Fig. 6.17B), are used where preservation of small form factor is important. They find applications in settop boxes and ultrathin laptops, but typically require a properly designed system board that is equipped with the correct connector and allows sufficient installation space.

A companion specification to SATA is the Advanced Host Controller Interface (AHCI) developed by Intel (currently at revision 1.3.1). It describes an implementation-independent, register-level interface between the host controller hardware and system software. The specification allows system programmers to support correctly additional hardware features such as NCQ and hot-swapping of I/O devices. AHCI is supported by default by many popular operating systems, such as Windows, Mac OS, and Linux.

6.7.3 JTAG

JTAG is a low-level hardware interface specified by IEEE Standard 1149 [11]. It takes its name from the Joint Test Action Group, which in the mid-1980s set out to develop verification and test methods for electronic circuits. While most casual computer users are never likely to have an opportunity to use JTAG directly, it is broadly adopted by the industry for postproduction printed circuit board testing (detection of shorts, mismatched and detached pins, "stuck" bits, in-silicon logic defects, and so on). As the density of integrated circuits increased, it quickly became uneconomical to provide explicit test points on board for all supported features. Additionally, JTAG permits in-circuit debugging of embedded applications by being able to access most if not all of the device register state, including the status of I/O pins. Coupled with the built-in self-test functionality commonly implemented by manufacturers in most large-scale integration and VLSI logic circuits, JTAG may identify many chip failure modes before allowing them to enter the supply chain. Since it can directly manipulate the device hardware state, JTAG is also occasionally used to perform firmware updates in cases when more user-friendly options may not be available or desirable.

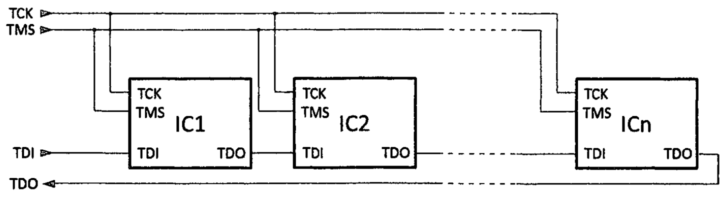


FIGURE 6.18

JTAG chain with n devices. Integrated circuit (IC) blocks represent individual integrated circuits that may be located on single or multiple printed circuit boards.

JTAG functionality relies on the presence of four signals: TCK (test clock), TDI (test data in), TDO (test data out), and TMS (test mode select). Optionally the interface may also contain a TRST (test reset) signal to perform reset of the test logic. To lower the required pin count, multiple JTAG-equipped devices may be daisy-chained as illustrated in Fig. 6.18. The test data and instructions are clocked in serially through the TDI line and output via TDO at the rising edge of the TCK (clock) signal. Beyond a few standard mandated exceptions, instruction semantics are implementation dependent. The level of the TMS pin influences the performed control function depending on the internal control state. Neither the JTAG connector nor its clock frequency is standardized; the latter may range from single to multiple 10s of MHz. The host may enable a bypass operation in any device on a chain, thus avoiding full communication with it if not required. A variant of JTAG permits a two-wire interface using only the TCK and TMSC (test serial data) signals, as described by the IEEE 1149.7 revision of the standard. The update addresses one of the common problems of the daisy-chained JTAG: to operate, all devices in the chain have to be powered up. IEEE 1149.7 also permits a star topology to be realized.

All JTAG implementations must support the test access port (TAP) with the TCK, TMS, TDI, and TDO pins, TAP controller, at least a two-bit wide instruction register, a one-bit bypass register, and a boundary scan register (one bit or more). Optionally, a 32-bit long IDCode register may also be exposed so that individual devices in the chain may be identified by the host. The instruction and data registers form parallel data paths that share the data input TDI as well as the output TDO. The TAP controller embeds a predefined state machine with 16 states. The transitions between states are performed in accordance with the TMS value during active clock edge. Individual states may force normal operation, invoke test functions defined by the contents of the instruction register, pause testing, and perform capture, update, or logical shift of instruction or data register contents. The required instructions for all implementations include device bypass and boundary scan support (state sampling, register load, and internal and external test execution).

Practical implementations frequently extend the basic JTAG instruction set and test range using custom chip-specific logic. For this reason vendors provide specialized software tools (command line and GUI based) that directly support native capabilities of the implementation without requiring the hardware engineer or system programmer to be familiar with the low-level details.

6.7.4 UNIVERSAL SERIAL BUS

Besides high performance components, computers need to communicate with relatively low-speed attached peripheral devices such as keyboards and printers. While these needs have been addressed in the past by a number of both specialized (such as IBM's PS/2 connector for mouse and keyboard) and industry-standard (e.g., serial and parallel communication ports) interfaces, their usefulness was limited when new types of attached devices became available. Among the shortcomings of previous solutions were bulky connectors, reduced interchangeability and interoperability, lack of an option to provide power to the attached devices, minimal or no ability to retrieve the type and operating parameters of connected peripherals, limited support for automatic configuration, no straightforward way to expand the number of available access ports, and, in some cases, insufficient communication speeds.

The USB standard [12] introduced in the mid-1990s successfully resolved these issues. It is currently guided by USB Implementers Forum Inc., a nonprofit corporation involving representatives of 894 hardware and software companies, including among others Intel, Hewlett–Packard, NEC, Renesas, Samsung, ST Microelectronics, Infineon, Philips, Sony, Apple, and Microsoft. The standard has been designed with low cost and simplicity as the primary features. The USB standard defines the architecture, data-flow model, mechanical and electrical properties of connectors and cables, signaling and physical layer, power supply and management, and transaction protocols. It is currently implemented in many categories of peripheral devices, including keyboards, mice, printers, scanners, cameras, mobile phones, media players, mass storage, modems, network adapters, game controllers, and more. The standard underwent updates in three major revisions that successively increased the communication bandwidth, detailed new connector types, defined multihost communication mode (USB On-The-Go), and specified additional power management and battery-charging protocols. The most recent version 3.1 was released in 2013.

USB provides a bidirectional communication link that originally operated at 1.5 Mbps ("low-speed") and 12 Mbps ("full-speed") signaling rates. Due to only one differential pair of wires dedicated to data transfer (the other two pins being ground and +5 V supply rail), the communication only supported half-duplex mode. The USB 2.0 update in 2000 increased the raw data rate to 480 Mbps ("hi-speed" mode), but due to protocol overheads the sustained data rate achieved was only 25–40 MB/s, i.e., less than 70% of the peak. USB 3.0 introduced "superspeed" of up to 5 Gbps, signified by blue-colored receptacles. Some USB 3.0 connectors are backwards compatible with USB 2.0, and the increased data rates are achieved by using an additional four pins (two differential line pairs for transmit and receive, thus permitting full-duplex operation) on the opposite side of the connector. In USB 3.1, doubling the maximal signal rate to 10 Gbps required introduction of an entirely new connector format, Type-C. An overview of various USB connectors and receptacles is presented in Fig. 6.19. The specification limits the cable length of low-speed devices to 5 m, full-speed devices to 3 m, and 5 m for hi-speed devices. USB 3.0 currently does not impose cable length constraints.

USB uses a tiered star topology with a single host at the top level, shown in Fig. 6.20. To overcome the limited number of host ports, multiple hubs may be inserted to add additional tier levels for up to seven tiers total and a maximum of 127 USB devices. Functionally, each USB device conforms to the same organizational scheme. Individual logical subdevices are called *functions* and communicate with the host via *pipes*. Pipes are logical channels that connect the host with an *endpoint* of a specific subdevice. A maximum of 16 input and 16 output endpoints per device are permitted. Endpoints are initialized in a process called *enumeration* (performed right after device

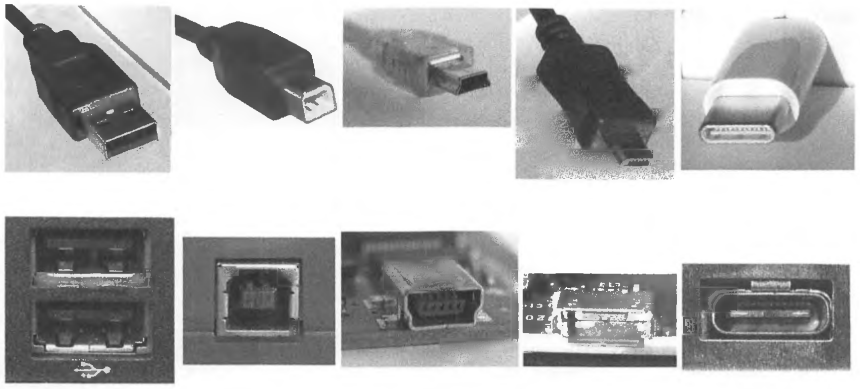


FIGURE 6.19

Comparison of common USB connector types (plugs on top, receptacles on bottom).

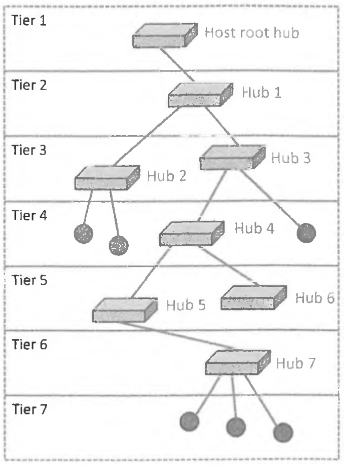


FIGURE 6.20

USB devices connected in multitiered topology.

power-up) and stay assigned as long as the device is connected. Pipes convey messages and streams: the first are relatively short commands that generate status response, while the latter are unidirectional and support *isochronous* (repeated communication with guaranteed bandwidth), *interrupt* (bounded latency communication), or *bulk* (asynchronous communication that may use the remaining link bandwidth) transfers. USB distinguishes several device classes, such as printer or mass storage, to facilitate loading the appropriate driver software on the host. Common features shared by devices in the same class are frequently supported by the operating system without the need for device-specific manufacturer software.

USB makes provisions for powering or charging peripheral devices through the same socket that facilitates the data transfer. Separate 5 V power and ground lines are included for that purpose. Nominally the current draw has to be negotiated with the host, and is limited to 100 mA for low-power devices (0.5 W) and 500 mA for high-power devices (2.5 W). These limits have been raised for superspeed devices to 150 and 900 mA, respectively. Unfortunately, not all devices comply with the specifications and draw more current than permitted, which on occasion may lead to their erratic behavior. The proliferation of mobile gadgets prompted the creation of a new port type, a *charging port*, which must deliver at least 1.5 A of current. It exists in a version that supports data communication (*charging downstream port*) and one that does not (*dedicated charging port*). A separate specification, *USB Power Delivery*, intends to extend this support to other devices such as laptops or hard drives by providing six power profiles to supply up to 100 W at three voltage levels using a dedicated power configuration protocol.

6.8 SUMMARY AND OUTCOMES OF CHAPTER 6

- The most widely used form of a high performance computer is the symmetric multiprocessor (SMP) architecture.
- SMPs are also referred to as *shared-memory* (SM) machines or *cache-coherent* computers.
- SMP architecture integrates a number of processor cores with a single shared main memory system by means of a common interconnection network.
- The symmetric multiprocessing attribute requires that copies of main memory data values that are held in caches for fast access must be consistent (cache coherency).
- Every SMP has multiple I/O channels that communicate with external devices (outside the SMP), user interfaces, data storage, system area networks, local area networks, and wide area networks, among others.
- A fundamental consequence of Amdahl's law is that independent of the size of the accelerator's peak performance gain, g, the sustained performance is bounded by the fraction, f, of the original code that can be accelerated.
- Defining characteristics of processors include the number of cores per socket, the size and interconnectivity of the cache levels (usually two or three levels), the clock rate of the core, the number and type of arithmetic logic units per core (ILP), the die size (between one and four square centimeters), the feature size, and the delivered performance for one or more standardized benchmarks.
- The pipeline logic structure is a general way of exploiting a form of very fine-grain parallelism, as each of the pipeline stages is operating simultaneously.

- Multithreading incorporates multiple instruction streams or threads, through sets of multiple instruction pointers and their associated register sets.
- The "memory wall" recognizes the mismatch between the peak demand rate of the processor socket for data access and the possible delivered throughput and latency of the main memory technology, principally semiconductor DRAM.
- The memory hierarchy or stack consists of layers of memory storage technology, each with different tradeoffs between memory capacity, costs, and cycle times which reflect bandwidths and latencies.
- In a modern SMP the cache is usually not a single layer of higher-speed memory but rather multiple layers to create an optimal balance of speed and size.
- PCI Express is a dominant standard for attaching and high performance communication with expansion boards on different machines that may use processor architecture other than the original Intel ×86 variant.
- The two most ubiquitous network interface controllers are Gigabit Ethernet and IB.
- SATA is used primarily to provide connectivity to mass-storage devices.
- JTAG is broadly adopted by the industry for postproduction printed circuit board testing.
- The USB standard provides a relatively low-speed method for communication with attached peripheral devices.

6.9 QUESTIONS AND EXERCISES

1. List and describe the components of an SMP node. Where applicable, name their most significant operational parameters and the units in which they are measured. Give approximate values these parameters may assume in common server hardware.
2. Expand and define each of the following acronyms. What are their application domains?
   - SMP
   - ILP
   - CPI
   - MAC
   - PCIe
   - SATA
   - USB
3. Why is standardization of I/O and expansion buses important? Provide examples.
4. You are an IT specialist at a small computational research institution. The scientists require a peak 100 Tflops machine to conduct their studies. The approved vendor offers two units rack-mount nodes with two CPU sockets that may accommodate either 12-core processors clocked at 3.4 GHz or 20-core processors operating at 2.5 GHz frequency. Each core can perform four floating-point operations per clock cycle. Given that there is 32U space for nodes available in each rack, answer the following.
   a. Which type of processor would you recommend to minimize the floor space occupied by racks?
   b. How many racks are needed to reach the required peak throughput?
   c. With all racks filled, what is the final peak computational throughput of the machine?

5. A sequential version of a simulation takes 90 min to compute 10,000 iterations. Each iteration can be accelerated using multiple threads of execution, but the overhead of assigning work to the threads is 100 ms. If the sequential setup of the application takes 10 min regardless of the number of iterations subsequently executed, how many cores are needed to bring the execution of a program performing 1000 iterations down to 12 min? What maximum speedup is possible assuming unlimited execution resources?
6. Execution of a 1,000,000-instruction program takes 2.5 ms on a 2.5 GHz core. The hardware monitor reports a cache miss ratio of 6% for the application. Main memory access takes on average 80 ns, while cache access has a latency of 800 ps. Given that all ALU instructions are executed effectively in a single clock cycle, calculate the following.
   a. The fraction of application instructions that performed ALU operations.
   b. If the core has a 16 KB cache and doubling the cache size decreases the miss rate by 1% for that particular application, what would be the required cache size (in powers of 2) to cut the execution time in half?
   c. What would the program runtime and resulting speedup be if all accessed data fits in the cache?

REFERENCES

[1] M.S. Papamarcos, J.H. Patel, A low-overhead coherence solution for multiprocessors with private cache memories, in: ISCA '84 Proceedings of the 11th Annual International Symposium on Computer Architecture, 1984.
[2] J. Wu, J.R. Larus, Static branch frequency and program profile analysis, in: MICRO 27 Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994.
[3] T.-Y. Yeh, Y.N. Patt, Two-level adaptive training branch prediction, in: MICRO 24 Proceedings of the 24th Annual International Symposium on Microarchitecture, 1991.
[4] C.-C. Lee, I.-C.K. Chen, T.N. Mudge, The bi-mode branch predictor, in: MICRO 30 Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, 1997.
[5] S. McFarling, Combining Branch Predictors, 1993. WRL Technical Note TN-36.
[6] D.A. Jimenez, C. Lin, Neural methods for dynamic branch prediction, ACM Transactions on Computer Systems (TOCS) 20 (4) (2002) 369–397.
[7] PCI-sig Specifications, 2017 [Online]. Available: http://pcisig.com/specifications/.
[8] R.M. Metcalfe, D.R. Boggs, Ethernet: distributed packet switching for local computer networks, Communications of the ACM 19 (7) (1976) 395–404.
[9] TOP500 Highlights – June 2017, 2017 [Online]. Available: https://www.top500.org/lists/2017/06/highlights/.
[10] OpenFabrics Alliance, 2017 [Online]. Available: https://www.openfabrics.org/.
[11] IEEE Standards Association, 1149.1-2013-IEEE Standard for Test Access Port and Boundary-scan Architecture, 2013 [Online]. Available: http://standards.ieee.org/findstds/standard/1149.1-2013.html.
[12] USB-IF, Universal Serial Bus Revision 3.1 Specification, July 26, 2013 [Online]. Available: http://www.usb.org/developers/docs/usb_31_062717.zip.

CHAPTER

THE ESSENTIAL OPENMP 7

CHAPTER OUTLINE

7.1 Introduction **225**
7.2 Overview of OpenMP Programming Model **226**
7.2.1 Thread Parallelism 226
7.2.2 Thread Variables 228
7.2.3 Runtime Library and Environment Variables 228
7.2.3.1 Environment Variables 229
7.2.3.2 Runtime Library Routines 230
7.2.3.3 Directives 230
7.3 Parallel Threads and Loops **231**
7.3.1 Parallel Threads 231
7.3.2 Private 232
7.3.3 Parallel "For" 233
7.3.4 Sections 239
7.4 Synchronization **241**
7.4.1 Critical Synchronization Directive 242
7.4.2 The Master Directive 242
7.4.3 The Barrier Directive 243
7.4.4 The Single Directive 243
7.5 Reduction **244**
7.6 Summary and Outcomes of Chapter 7 .... **245**
7.7 Questions and Problems **246**
Reference **247**

7.1 INTRODUCTION

OpenMP is an application programming interface (API) to support the shared-memory multiple-thread form of parallel application development. "OpenMP" stands for "open multiprocessing" [1]. It greatly simplifies the development of multiple-threaded parallel programming compared to, for example, low-level operating system (OS) support services for threads and shared memory. OpenMP incorporates separate sets of bindings for the sequential programming languages Fortran, C, and C++. It gives easy access to the resources of the general class of symmetric multiprocessor (SMP) computers for parallel applications.

This chapter describes the syntax constructs related to the C programming language bindings. For those unfamiliar with C programming, a tutorial, "The Essential C", is offered in the appendix of this textbook. OpenMP provides extensions to C in the form of compiler directives, environment variables, and runtime library routines to expose and execute parallel threads in the context of shared memory. A principle of the design philosophy is to permit incremental changes to sequential C code for ease of use and natural migration from initial C-based applications to parallel programs. In so doing it provides a practical and powerful means of parallel computing, admittedly within the scalability limits of the SMP class of parallel computers.

OpenMP is among the most widely used parallel programming APIs. However, it is limited in scalability to hardware system architectures providing near uniform memory access (UMA) to shared memory. This chapter provides an introductory treatment to the essentials of OpenMP and how to program in parallel with it. Although it is an initial coverage assuming no prior experience with parallel programming, this chapter provides all the necessary concepts and semantic constructs to enable the development of useful real-world applications. This presentation is primarily focused on release 2.5, which is core to later releases and is probably the most widely used version as well as most broadly supported.

Shared-memory parallel architectures were first developed in the 1980s and a number of APIs were devised to assist in programming these. The OpenMP specification standard process began in 1997 based on this prior work and an early draft of such an interface, ANSI X3H5, was released in 1994. OpenMP evolution is overseen by the OpenMP Architecture Review Board, consisting of industry and government partners. The first C-based specification, C/C++ 1.0, was released in 1998, followed by C/C++ 2.0 in 2002. C and Fortran specifications have been released together since 2005, with OpenMP 2.5, 3.0, and 3.1 released in 2005, 2008, and 2011, respectively. The most recent version 4.5, released in 2015.

7.2 OVERVIEW OF OPENMP PROGRAMMING MODEL

OpenMP provides a shared-memory multiple-threads programming model. It assumes underlying hardware support for efficient management of shared memory, including virtual addresses and cache coherency among processor cores and across multiple sockets. This is the principal defining facet of the SMP class of parallel computers. All processor cores have direct access to all memory shared within the system. A simple but illustrative representation of the class of parallel computers suitable for OpenMP programming is shown in Fig. 7.1. The key elements are the processor cores, P, that perform the concurrent threaded computing, the memory banks, M, that are equally accessible by the threads, and the connectivity between both P and M elements that enables the shared-memory architecture and execution models.

7.2.1 THREAD PARALLELISM

Threads are the principal means of providing parallelism of computation. A thread is an independently schedulable sequence of instructions combined with its private variables and internal control. Usually there are as many threads allocated to the user computation as there are processor cores assigned to the computation. However, this is not required. Threads are divided among the master

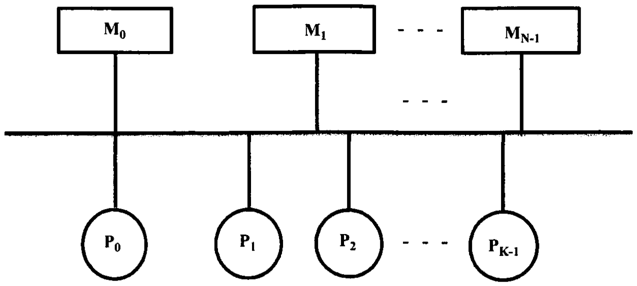


FIGURE 7.1

Shared-memory multiprocessor.

thread and the worker threads. The single master thread exists for the lifetime of the computation, from its initiation to its termination. Sometimes the master thread is the only thread being performed at one time. The worker threads provide additional paths of concurrency of execution for performance gain. Worker threads are controlled by the master thread and are delineated by OpenMP directives. Like the number of threads, how the threads are scheduled is determined by environment variables and can be static or dynamic.

OpenMP supports the fork-join model of parallel computing. At particular points in the execution the master thread spawns a number of threads and with them performs part of the program in parallel. The point of initiation of multiple worker threads is referred to as the *fork*. Usually all these threads perform their calculations separately, and when they reach completion they wait for the other threads to finish as well. This is the *join* of the parallel threads, and the default at the join is an implicit barrier synchronization. All the threads must complete before computation is continued beyond that point of control. An OpenMP parallel program mostly consists of a sequence of such fork–join worker and master thread parallel segments separated by lone sequential master thread segments, as shown in Fig. 7.2. Segments of concurrent master/worker threads often have all the threads the same, differentiated only by the values of their private variables. This is the single-program, multiple data (SPMD) model. Alternatively, the concurrent threads may each execute different code blocks, separately delineated by appropriate directives. In either case, join synchronization at the end of the concurrent threads is enforced unless explicitly avoided through added directives for this purpose. The figure illustrates this parallel control flow. The horizontal axis represents time from left to right, while the vertical access shows work in terms of one or more concurrent threads. The single lowest line is the master thread that continues from the beginning to the end of the OpenMP program. At key fork points multiple threads are launched for work that can be performed concurrently. These threads may be somewhat irregular, in that they do not execute exactly the same work even if their code is the same. When all the concurrent threads have completed at the join synchronization point, the computation can proceed; in each case by the master thread alone until the next thread fork is encountered.

OpenMP permits the representation of nested parallelism, such that inner fork–join segments of parallel threads can themselves be embedded into threads of outer parallel thread segments. However, while the syntax is supported and will execute correctly, not all implementations will take advantage of this additional parallelism and may treat it as sequential code, one inner thread after another. An

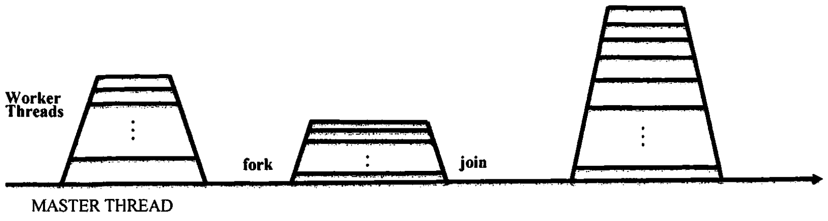


FIGURE 7.2

Fork–join model of master/worker threads.

example of nested parallelism is illustrated in Fig. 7.3. Again, the lower horizontal line represents the master thread, with time increasing from left to right. First a set of worker threads is created when the master thread encounters a forking point of parallelism; this is the *outer fork.* Each of these outer threads then separately encounters its own *inner fork* to create a second level of parallel threads, giving more concurrency for scalability. The inner threads of each outer thread then synchronize with their respective matching inner join, after which the outer thread proceeds until it encounters the outer join synchronization point with the other outer threads. The OpenMP scheduler uses this added parallelism to improve performance when possible.

7.2.2 THREAD VARIABLES

OpenMP is a shared-memory model allowing direct access to global variables by all threads of a user process. To support the SPMD modality of control where all concurrent threads run the same code block simultaneously, OpenMP also provides private variables. These have the same syntactical names, but their scoping is limited to the thread in which they are used. Private variables of the same name have different values in each thread in which they occur. A frequently occurring example is the use of index variables accessing elements of a vector or array. While all threads will use the same index variable name, typically "i", when accessing an element of the shared vector, perhaps "x[i]", the range of values of the index variable will differ for the separate concurrent threads. For this to be possible, the index variable has to be private rather than shared. In fact this particular idiom is so common that the default for such variable usage is private, although for most common variables the default is shared. Directive clauses are available for explicit setting of these properties of variables by users. Another variant on how variables may be used relates to reduction operators such as sum or product. In this specialized case the reduction variable is a mix of private and global, as discussed in detail in Section 7.5.

7.2.3 RUNTIME LIBRARY AND ENVIRONMENT VARIABLES

An OpenMP parallel application program consists of the syntax of the core language (i.e., Fortran, C, C++) with additional constructs to guide parallel threaded execution and set specific operational properties. These include environment variables, compiler directives, and runtime libraries. Environment variables define operational conditions and policies under which the executing OpenMP

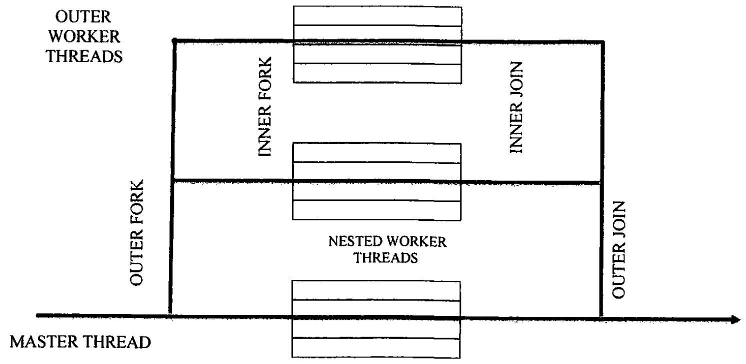


FIGURE 7.3

Nested parallel threads.

program will run. Their values can be set at the system shell through OS user interface commands; from within the program they can be accessed through runtime library routines. Compiler directives appear as comments, but through the OpenMP extensions, pragmas, are treated as commands to guide parallel execution. Additional functionality is provided through runtime library routines that help manage parallel programs. Many runtime library routines have corresponding environment variables that can be controlled by users. Examples include determining the number of threads and processors, scheduling policies to be used, and portable wallclock timing.

7.2.3.1 Environment Variables

OpenMP provides environment variables for controlling execution of parallel codes. These can be set from the OS command line or equivalent prior to execution of the application program, using export or setenv commands depending on the user shell. These variables have default settings, so they only have to be set explicitly if an optional value is required. There are four main environment variables.

OMP_NUM_THREADS controls the parallelism of the OpenMP application by specifying the number of threads to be used by the user program. This normally determines the number of cores allocated to the user program, but not always. When more threads are requested than available, they may be associated with OS Pthreads, requiring the OS to context switch adding additional overheads to the computing. The default option is system dependent. To set the value of this environment variable to 8 use the following bash command:

```
export OMP_NUM_THREADS=8
```

OMP_DYNAMIC enables dynamic adjustment of the number of threads for execution of parallel regions. Under certain conditions this provides a level of adaptability that makes optimal use of task granularity to minimize the effects of overhead and irregularity. However, it also incurs additional overhead within the runtime system and may not always improve performance. The default value for this environment variable is FALSE, which implies that the number of threads employed remains fixed at the value set in the environment variable OMP_NUM_THREADS. To change this to enable dynamic thread allocation, use the following bash command:

```
export OMP_DYNAMIC=TRUE
```

OMP_SCHEDULE manages the load distribution in loops such as the parallel for pragmas (discussed in Section 7.3.3). This environment variable sets the schedule type and chunk size for all such loops. The chunk size can be provided as an integer number. The default value for OMP_-SCHEDULE is 1. To set this two-tuple variable use the following form:

```
export OMP_SCHEDULE=schedule,chunk
```

OMP_NESTED permits nested parallelism in OpenMP applications. This may give the opportunity for more parallelism, which may increase scalability, but at the risk of finer granularity in the presence of fixed overheads, which may reduce efficiency. The default value for this Boolean environment value is FALSE, permitting only the top level of fork–join parallelism to be used. To set this environment variable to support multilevel parallelism, use the following form:

```
export OMP_NESTED=TRUE
```

7.2.3.2 Runtime Library Routines

Runtime library routines help manage parallel application execution, including accessing and using environment variables such as those above. Prior to using these routines in the code, the following *include* statement must be added:

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

Two important routines allow the program to know how many threads are operating concurrently and identify a unique rank for each thread among the total set. The first function,

```
omp_get_num_threads()
```

returns the total number of threads currently in the group executing the parallel block from where it is called. Usually, this is a direct reflection of the environment variable OMP_NUM_THREADS.

The second important runtime routine,

```
omp_get_thread_num()
```

returns a value to each thread executing the parallel code block that is unique to that thread and can be used as a kind of identifier in its calculations. When the master thread calls this function, the value of 0 is always returned, identifying its special role in the computation. A call to this routine by a worker thread will return a value between 1 and the environment variable value OMP_NUM_THREADS -1.

7.2.3.3 Directives

OpenMP directives are a principal class of constructs used to convert initially sequential codes incrementally to parallel programs. They serve a multitude of purposes, primarily about controlling parallelism through delineation and synchronization. The following sections describe in detail the directives for parallelism, mutual exclusion of shared variables through synchronization, and reduction calculations. All directives take the following form:

```
#pragma omp <directive> <clauses> <statement/code block>
```

As shown in the examples, such directives may be treated one at a time in a nested organization, or in many cases combined to simplify textual presentation. The clauses permit optional conditions to be satisfied, like declaring the scope of a variable (e.g., private).

7.3 PARALLEL THREADS AND LOOPS

7.3.1 PARALLEL THREADS

In the tradition of C programming and the teaching thereof, the first program to present is "Hello World". Using OpenMP, a very simple parallel program can be constructed to print this statement by multiple threads. It requires only one OpenMP command,

```
#pragma omp parallel
```

to turn the classic sequential C code to a parallel code. This simplicity of transformation from serial to parallel is one of the hallmark strengths of OpenMP. Parallel Hello World is written as follows.

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

int main() {
#pragma omp parallel
{
  printf("Hello World \n");
}

  return 0;
}
```

Code 7.1. Parallel Hello World example.

```
Hello World
Hello World
Hello World
Hello World
```

Output 1. The result from Code 7.1 when setting the environment OMP_NUM_THREADS to be 4.

That's it! When compiled and run, the result will be a succession of printed lines of text: Hello World. The number of such lines is determined by the environment variable OMP_THREAD_NUM.

The only difference between this and the conventional C version of this code is that in the parallel version is the single OpenMP directive: #pragma omp parallel is added. The effect of the parallel pragma is to fork the number of allocated threads, and for each such thread to execute the designated block of code. As a result every thread executes the printf statement once, each printing "Hello World". This admittedly rather useless code demonstrates the use and power of the parallel pragma, perhaps the most important directive in OpenMP. It creates the forks and subsequent joins of parallel threads throughout the program.

7.3.2 PRIVATE

In the simple version of Hello World, above, there is nothing to differentiate between the separate executed threads. For threads to become useful they need to support distinct work even if the code block is the same. Some local state for each thread is usually required, although there are some special cases (like reduction variables). The *private* clause within directives is the principal means of achieving this. It declares a variable within the code block to be local to each thread. By this means each thread has its own copy of the named variable, permitting each thread to have its own value, independent of the other threads executing the same code block.

```
#include <stdio.h>
#include <omp.h>
int main() {
  int num_threads, thread_id;
#pragma omp parallel private(num_threads, thread_id)
{
    thread_id = omp_get_thread_num();
    printf("Hello World. This thread is: %d\n", thread_id);
    if (thread_id == 0) {
    num_threads = omp_get_num_threads();
    printf("Total # of threads is: %d\n", num_threads);
    }
}
   return 0;
}
```

Code 7.2. An example hello world code where each OMP thread prints out its thread id.

```
Hello World. This thread is: 0
Total # of threads is: 4
Hello World. This thread is: 3
Hello World. This thread is: 1
Hello World. This thread is: 2
```

Output 2. The output from Code 7.2 with the environment OMP_NUM_THREADS set to be 4.

There are two variables, num_threads and thread_id, that are declared as private, and therefore separate copies of both are provided for each and every thread with the values potentially different. In this case num_threads is only used by the master thread, thread 0. The omp_get_num_threads runtime function is called only by the master thread. Each thread has its own copy of thread_id. The runtime function omp_get_thread_num() will return a different and unique value to each thread for the respective version of thread-id.

7.3.3 PARALLEL "FOR"

Among the most useful sources of parallelism is the distribution of loops among threads. In C, loops are defined by the "for" construct specifying the number of iterations of a code block through a designated range of an index variable local to that loop code. Here is a sequential code block to add two arrays together.

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

int main (int argc, char *argv[])
{
  const int N = 20;
  int nthreads, threadid, i;
  double a[N], b[N], result[N];

  // Initialize
  for (i=0; i < N; i++) {
    a[i] = 1.0*i;
    b[i] = 2.0*i;
  }

  for (i=0; i<N; i++) {
    result[i] = a[i] + b[i];
  }

   printf(" TEST result[19] = %g\n",result[19]);

   return 0;
}
```

Code 7.3. Serial example of adding two arrays together.

```
TEST result[19] = 57
```

Output 3. The output from Code 7.3.

This program has three parts, like many real-world programs: initialization, calculation, and output. In this simple program most of the lines of text are dedicated to the declaration and initialization of program variables. Here these include integers N, nthreads, threadid, and i, as well as the double vectors a, b, and result. A for-loop is included to initialize the vector elements to (admittedly gratuitous) double values. The output of the computation is given by the single printf statement that sends the last element of the result vector to standard inout/output (usually the screen or a file).

This program can be readily converted to be executed in parallel to increase computational performance and reduce the time to solution. Three additions are required to transform the above program to parallel.

1. Include the OpenMP header.
2. Delineate the code block to be made parallel.
3. Specify the loop to be distributed among the concurrent threads.

The first two of these semantic constructs have already been discussed. The OpenMP libraries are incorporated through the command:

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

The parallel code block is established through the directive:

```
#pragma omp parallel
{
...
}
```

The new construct is the `parallel for` instruction. This directive enables work sharing, where the total work of a loop is divided among the assigned threads. The effect is to divide the range of the private index variable of the for-loop into subranges, preferably of equal span, with one assigned to each of the parallel threads. Thus each thread is responsible for part of the total work of the loop; with all the threads working on their respective parts separately but at the same time, parallel computing is performed. Optimally, the speedup would be equal to the number of threads being used, but for a number of reasons (discussed later) it is infrequent that this level of scaling is fully realized. The `parallel for` directive is given as the following:

```
#pragma omp for
```

The parallel version of the vector addition example is presented in Code 7.4 with the additional OpenMP directive in lines 1, 17, and 20:

```
 1 #include <omp.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 int main (int argc, char *argv[])
 6 {
 7   const int N = 20;
 8   int i;
 9   double a[N], b[N], result[N];
10
11   // Initialize
12   for (i=0; i < N; i++) {
13     a[i] = 1.0*i;
14     b[i] = 2.0*i;
15   }
16
17 #pragma omp parallel
18   { // fork
19
20   #pragma omp for
21   for (i=0; i<N; i++) {
22     result[i] = a[i] + b[i];
23   }
24
25   } // join
26
27   printf(" TEST result[19] = %g\n",result[19]);
28
29   return 0;
30 }
```

Code 7.4. OpenMP parallel for version of Code 7.3. OpenMP additions are seen in lines 1, 17, and 20. The output of Code 7.4 is the same as Code 7.3, shown in Output 3.

While correct, the code above is a bit verbose. OpenMP allows some compression of code text by merging different directives where it makes sense. For example, the parallel and for directives can be combined into a single statement, as shown in Code 7.5:

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

int main (int argc, char *argv[])
{
  const int N = 20;
  int i;
  double a[N], b[N], result[N];

  // Initialize
  for (i=0; i < N; i++) {
    a[i] = 1.0*i;
    b[i] = 2.0*i;
  }

  #pragma omp parallel for
  for (i=0; i<N; i++) {
  result[i] = a[i] + b[i];
  }

  printf(" TEST result[19] = %g\n",result[19]);

  return 0;
}
```

Code 7.5. Combining the parallel and for directives into a single statement in Code 7.4

Notice that the braces are not required because a single statement now makes up the code block. To find out which thread executes which index of the vector addition, some additional statements are needed, as shown in Code 7.6.

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

int main (int argc, char *argv[])
{
  const int N = 20;
  int nthreads, threadid, i;
  double a[N], b[N], result[N];

  // Initialize
  for (i=0; i < N; i++) {
    a[i] = 1.0*i;
    b[i] = 2.0*i;
  }

#pragma omp parallel private(threadid)
  { // fork
  threadid = omp_get_thread_num();

  #pragma omp for
  for (i=0; i<N; i++) {
    result[i] = a[i] + b[i];
    printf(" Thread id: %d working on index %d\n",threadid,i);
  }

  } // join

  printf(" TEST result[19] = %g\n",result[19]);

  return 0;
}
```

Code 7.6. In this example of vector addition, the identity of the thread performing the operation for each index is printed to screen.

In Code 7.6 a new variable is introduced: threadid. It contains the OpenMP thread index and is initialized inside the OpenMP parallel region. Because it is declared outside the scope of the parallel region, it would by default be considered a global variable by OpenMP. Thus it is necessary to declare it as private in the clause following the OpenMP parallel pragma in line 17.

```
Thread id: 0 working on index 0
Thread id: 0 working on index 1
Thread id: 0 working on index 2
Thread id: 0 working on index 3
Thread id: 0 working on index 4
Thread id: 0 working on index 5
Thread id: 0 working on index 6
Thread id: 1 working on index 7
Thread id: 1 working on index 8
Thread id: 1 working on index 9
Thread id: 1 working on index 10
Thread id: 1 working on index 11
Thread id: 1 working on index 12
Thread id: 1 working on index 13
Thread id: 2 working on index 14
Thread id: 2 working on index 15
Thread id: 2 working on index 16
Thread id: 2 working on index 17
Thread id: 2 working on index 18
Thread id: 2 working on index 19
TEST result[19] = 57
```

Output 4. The output from Code 7.6 when using OMP_NUM_THREADS=3. The default thread scheduler in OpenMP will break the for-loop roughly into three equal pieces: thread 0 works on array indices 0 through 6, thread 1 works on array indices 7 through 13, and thread 2 works on array indices 14 through 19.

The behavior controlling which thread works on which index can be altered by using the schedule clause, as noted in Section 7.2.3.1. This is illustrated in Code 7.7.

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

int main (int argc, char *argv[])
{
  const int N = 20;
  int nthreads, threadid, i;
  double a[N], b[N], result[N];

  // Initialize
  for (i=0; i < N; i++) {
    a[i] = 1.0*i;
    b[i] = 2.0*i;
  }

```

```
   int chunk = 5;

#pragma omp parallel private(threadid)
    { // fork
    threadid = omp_get_thread_num();

  #pragma omp for schedule(static,chunk)
  for (i=0; i<N; i++) {
      result[i] = a[i] + b[i];
      printf(" Thread id: %d working on index %d\n",threadid,i);
  }

  } // join

  printf(" TEST result[19] = %g\n",result[19]);

  return 0;
}
```

Code 7.7. An example of the schedule clause. Work in the for-loop will be statically divided into chunks of size 5.

```
Thread id: 0 working on index 0
Thread id: 0 working on index 1
Thread id: 0 working on index 2
Thread id: 0 working on index 3
Thread id: 0 working on index 4
Thread id: 0 working on index 15
Thread id: 0 working on index 16
Thread id: 0 working on index 17
Thread id: 0 working on index 18
Thread id: 0 working on index 19
Thread id: 1 working on index 5
Thread id: 1 working on index 6
Thread id: 1 working on index 7
Thread id: 1 working on index 8
Thread id: 1 working on index 9
Thread id: 2 working on index 10
Thread id: 2 working on index 11
Thread id: 2 working on index 12
Thread id: 2 working on index 13
Thread id: 2 working on index 14
TEST result[19] = 57
```

Output 5. Output from Code 7.7 when run using OMP_NUM_THREADS=3. The for-loop is statically divided into chunks of size 5 among three threads. Hence thread 0 operates on array indices 0—4 and 15—19, thread 1 operates on array indices 5—9, and thread 2 operates on array indices 10—14.

7.3.4 SECTIONS

OpenMP provides a second powerful method for specifying work sharing among parallel code blocks. The `sections` directive describes separate code blocks, each containing a different sequence of instructions, which may be performed concurrently. One thread is allocated to each code block. The full set of parallel blocks is initiated with the following directive:

```
#pragma omp sections
{ ... }
```

Within this structure is the set of nested code blocks, each begun by the directive:

```
#pragma omp section
{ <code block> }
```

with the exception of the first code block that does not require its own sections pragma (the sections pragma serves this second duty) heading. A simple example of a sections code block structure could look like this:

```
#pragma omp parallel
{
#pragma omp sections
 {
  {
   <1st parallel code block>
  }
 #pragma omp section
  {
   <2nd parallel code block>
  }
 #pragma omp section
  {
   <3rd parallel code block>
  }
 }
}
```

This nested structure of code blocks can be extended to represent as many distinct and concurrent blocks as necessary. But depending on the number of threads specified by the environment variable, not all of these may be executed simultaneously.

The example in Code 8 demonstrates the use of the sections and nested section directives to specify three separate code blocks to be executed concurrently. The three calculations determine statistics about a set of integer values, x. The first determines the minimum and maximum values of the set. The second computes the mean. The third computes the mean of the square of the values, which is used later to provide the variance.

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

```

int main()
{
  const int N = 100;
  int x[N], i, sum,sum2;
  int upper, lower;
  int divide = 20;
  sum = 0;
  sum2 = 0;

#pragma omp parallel for
  for(i = 0; i < N; i++) {
    x[i] = i;
  }


#pragma omp parallel private(i) shared(x)
{

// Fork several different threads
#pragma omp sections
  {
      {
        for(i = 0; i < N; i++) {
          if (x[i] > divide) upper++;
          if (x[i] <= divide) lower++;
        }
        printf("The number of points at or below %d in x is %d\n",divide,lower);
        printf("The number of points above %d in x is %d\n",divide,upper);
      }
#pragma omp section
      { // Calculate the sum of x
        for(i = 0; i < N; i++)
          sum = sum + x[i];
        printf("Sum of x = %d\n",sum);
      }
#pragma omp section
      {
        // Calculate the sum of the squares of x
        for(i = 0; i < N; i++)
          sum2 = sum2 + x[i]*x[i];

        printf("Sum2 of x = %d\n",sum2);
      }
  }
}
  return 0;
}
```

Code 7.8. **Example of sections in OpenMP.**

7.4 SYNCHRONIZATION

One strength of OpenMP is the sharing of global data among multiple concurrent threads. This "shared-memory" model presents a view of program data similar to that experienced with the use of conventional sequential programming interfaces like the C language. This is distinguished from "distributed-memory" models where special send–receive message-passing semantics are required to exchange values among concurrent processes, such as found in the message-passing interface programming libraries. But with this ease of use comes a serious challenge: control of the order of access to shared variables. This problem in a different form was encountered when the distinction between private and shared variables was made. By designating a variable as private, it was possible to avoid the out-of-order problem among multiple threads; here, copies of a named variable disassociated the accesses of separate threads. However, communication between or among threads through shared memory is a frequent and efficient means of computation cooperation, if appropriately coordinated. OpenMP incorporates semantic constructs to enable coordination in the shared use of global memory for the class of SMP parallel architectures.

On a shared-memory system communication between threads is mainly through read and write operations to shared variables. Where two threads are both reading the value of a shared variable, previously set, the order of accessing the variable by the threads is irrelevant; either thread can perform the read first, followed by the second thread. But if one thread is responsible for setting the value through a global write for the other thread to read and use, then clearly the order of access is important and failing to ensure proper order will likely result in an error. This can become more complex when more than two threads are involved.

Synchronization defines the mechanisms to help in coordinating execution of multiple parallel threads that use a shared context (shared memory) in a parallel program. Without synchronization, multiple threads accessing a shared-memory location may cause conflicts. This can occur by two or more threads attempting to modify the same location concurrently. It can also happen if one thread is attempting to read a memory location while another thread is updating the same location. Without strict control of ordering, a race condition may make the result of these actions nondeterminant; the result cannot be guaranteed always to produce the same answer. Synchronization helps to prevent such races and access conflicts by providing explicit coordination among multiple threads. These include implicit event synchronization and explicit protection synchronization directives.

Implicit synchronization determines the occurrence of an event across multiple threads. Barriers are a simple form of event synchronization in OpenMP to coordinate multiple threads such that they are aligned in time. A barrier establishes a point in a parallel program where each thread waits for all the other like threads to reach the same point in their respective execution. This ensures that all the computing threads have completed their computation prior to that specific instruction. Only after all threads have reached the barrier can any of them proceed.

Explicit synchronization directly controls access to a specific shared variable. This guarantees that access to the identified data location is limited to one thread at a time. This is particularly important when the thread needs to perform a compound atomic sequence of operations, such as a read–modify–write, on a data element without intrusion of another thread. While this does not fix the order of access, it does protect a variable until any one thread's activity associated with the variable has been completed without conflict. This class of synchronization constructs provides mutual exclusion.

7.4.1 CRITICAL SYNCHRONIZATION DIRECTIVE

The OpenMP pragma `critical` provides mutual exclusion for access to shared variables by multiple threads. It provides protection against race conditions and the minimum performance degradation in the case when all likely accesses to a given shared variable are from multiple concurrent threads of the same code sequence. The `critical` directive delineates a block of code that only one thread is permitted to execute at a time. Any global variable that is accessed within that sequence of instructions is protected from attempts by multiple concurrent executing threads of the same code block. Once one thread enters the critical region, the other threads have to wait until it has exited the region. The order in which the different threads perform the critical code block is undetermined; only the limit of one thread at a time entering and completing the specified code is guaranteed. The `critical` pragma permits atomic read—modify—write operation sequences to be safely conducted on a shared variable.

The `critical` pragma has the form:

```
#pragma omp critical
{
...
}
```

An example of its use to perform compound atomic operations safely is the following:

```
int n;
n = 0;
...
#pragma omp parallel shared(n)
{
 #pragma omp critical /* delineate critical region */
  n = n + 1; /* increment n atomically */
} /* parallel end */
```

This simple code allows many threads to increment the shared variable n without the possibility of a race among them corrupting the resulting value. Independent of the order in which the critical regions of the separate threads are performed, the resulting value of n will be the same.

7.4.2 THE MASTER DIRECTIVE

The master directive provides another, and perhaps more simple, way of protecting a shared variable among threads to avoid races and possible corruption of result values. As the name implies, the directive gives total control to the master thread for a specified code block. Such a code block delineated by the master pragma is executed by only one thread, the master thread. When the master thread encounters the master directive, it proceeds to perform it like any other code. But when any thread other than the master thread (all the worker threads) reaches a master block, it does not execute it and skips over this part of the code. Thus this particular code block is only performed once, by the master thread. There is no possibility of a race condition because only one thread is allowed to access the global shared variables referred to within the master code block. There is no barrier implied by the master region. The worker threads that do not perform this code go right past it and continue without any delay caused by the master region. The master directive takes the following syntactical form:

```
#pragma omp master
{
 ... /* protected code block */
}
```

7.4.3 THE BARRIER DIRECTIVE

The barrier pragma puts the computation in a known control state. It synchronizes all the concurrent threads. When encountering a given barrier directive, all threads halt at that location in the code until all other threads have reached the same point of execution. Only when all the threads have reached the barrier can any of them proceed beyond it. Once all the threads have performed the barrier operation, they all continue with the computation after it.

The barrier operation is used to ensure that all the threads have completed the preceding computations no matter what order they are scheduled in or at what rate they are executing. An important purpose of this idiom is to implement the bulk synchronous parallel protocol, a very common form of parallel computation. With this approach, a set of threads reads from shared memory and performs the necessary arithmetic on their values. Then a barrier is performed. Only when all the threads have completed their computation and reached the barrier can they go ahead and write the resulting values back to the shared variables. In one form (there are several), after writing to shared memory every thread encounters a second barrier and again waits for all the other threads to complete their shared-memory write-back operation as well. Having safely performed all the writes, the threads can repeat the next step of the parallel calculation safely by reading the newly updated shared variables guaranteed to be correct because of the barriers. The barrier directive is:

```
#pragma omp barrier
```

7.4.4 THE SINGLE DIRECTIVE

The single directive combines a form of dynamic scheduling with synchronization. It expands the master pragma to permit any thread to perform the action, and combines this with an implicit barrier at the end. The delineated code block is executed by only one thread, like the master directive; but unlike master the executing thread can be any of the running threads, but only one of them. The first thread to reach the single pragma construct in its sequence of instructions will perform the designated code block. The remaining threads will not perform that code. But all the threads will encounter a barrier that blocks them from proceeding past the end of the single pragma code block until all of them have reached that point in their execution. Only after the thread executing the code designated by the single pragma has completed and exited that code can all the other threads continue. The single directive is:

```
#pragma omp single
{
 ... /* protected code executed by only one thread */
}
```

7.5 REDUCTION

Reduction operators are a means of bringing together a large number of values to produce a single result value. Familiar examples are numeric (integer or real) summation and logical OR over a range of variables. While this can be achieved through functions of more primitive operations, OpenMP (like other programming interfaces) provides a convenient way to accomplish reductions and in some cases to do so in parallel for performance speedup (over sequential implementations). The reduction pragma may take the following form:

```
#pragma omp reduction(op : result_variable)
{
 result_variable = result_variable op expression
}
```

The reduction operator, *op*, is one of the following:

```
+, *, -, /, &, ^, |
```

The *result_variable* is of a scalar value, with one such element as a private variable for every thread.

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

int main() {
  int i, n, chunk;
  float a[16], b[16], result;
  n = 16;
  chunk = 4;
  result = 0.0;


 for (i = 0; i < n; i++) {
   a[i] = i * 1.0;
   b[i] = i * 2.0;
 }

 #pragma omp parallel for default(shared) private(i) schedule(static, chunk) \
   reduction(+ : result)
   for (i=0; i < n; i++)
     result = result + (a[i] * b[i]);

 printf("Result = %f\n", result);
 return 0;
}
```

Code 7.9. Example of reduction.

```
Result = 2480.000000
```

Output 5. Output from Code 7.9.

7.6 SUMMARY AND OUTCOMES OF CHAPTER 7

- "OpenMP" stands for "open multiprocessing".
- OpenMP is an API for parallel computing that has bindings to programming languages such as Fortran and C.
- OpenMP supports programming of shared-memory multiprocessors, including SMP and distributed shared-memory classes of parallel computer systems.
- OpenMP supports the fork–join model of parallel computing. At particular points in the execution the master thread spawns a number of threads and with them performs a part of the program in parallel. The point of multiple worker thread initiation is referred to as the *fork*. Usually all these threads perform their calculations separately, and when they come to their respective completion they wait for the other threads to finish at the *join* of the parallel threads.
- OpenMP provides environment variables for controlling execution of parallel codes. These can be set from the OS command line or equivalent prior to execution of the application program.
- Runtime library routines help manage parallel application execution, including accessing and using environment variables such as those above. The library routines are provided in the omp.h file and must be included (`#include <omp.h>`) prior to using any of these routines.
- Threads are the principal means of providing parallelism of computation. A thread is an independently schedulable sequence of instructions combined with its private variables and internal control. Usually there are as many threads allocated to the user computation as there are processor cores assigned to the computation, although this is not required.
- `omp_get_num_threads()` returns the total number of threads currently in the group executing the parallel block from where it is called.
- `omp_get_thread_num()` returns a value to each thread executing the parallel code block that is unique to that thread and can be used as a kind of identifier in its calculations. When the master thread calls this function, the value of 0 is always returned, identifying its special role in the computation.
- OpenMP directives are a principal class of constructs used to convert initially sequential codes incrementally to parallel programs. They serve a multitude of purposes, primarily about controlling parallelism through delineation and synchronization.
- The parallel directive delineates a block of code that will be executed separately by each of the computing threads.
- The parallel for directive permits work sharing of an iterative loop among the executing threads, with one or more iterations performed by each thread.
- The private clause in a directive establishes that each thread has its own copy of a variable, and when accessing that designated variable will read or write its own private copy rather than a shared variable.
- The sections directive describes separate code blocks, each of a different sequence of instructions, which may be performed concurrently. There is one thread allocated to each code block.
- Synchronization directives define the mechanisms that help in coordinating execution of multiple parallel threads that use a shared context (shared memory) in a parallel program to preclude race conditions.

- The critical directive provides mutual exclusion of access to shared variables by permitting only one thread at a time to perform a given code block. When a thread enters the critical code section, all other threads that attempt to do so are deferred until the thread doing it has completed. Other threads are then free to execute the critical section of code themselves, but only one at a time.
- The master directive delineates a block of code that is only executed by the master thread, with all other threads skipping over it.
- The single directive delineates a block of code that is performed by only a single thread, but it can be any of the executing threads—whichever one gets to that code block first. All threads wait until the thread completing that code executes it.
- The barrier directive is a form of synchronization. When encountering a given barrier directive, all threads halt at that location in the code until all other threads have reached the same point of execution. Only when all the threads have reached the barrier can any of them proceed beyond it. Once all the threads have performed the barrier operation, they all continue with the computation after it.
- Reduction operators combine a large number of values to produce a single result value. A number of operations can be used for this purpose, such as + and | among others.

7.7 QUESTIONS AND PROBLEMS

1. Can you spot any mistakes in the following code? Please correct them.

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

// compute the dot product of two vectors

int main() {
  int const N=100;
  int i, k;
  double a[N], b[N];
  double dot_prod = 0.0;

  // Arbitrarily initialize vectors a and b
  for(i = 0; i < N; i++) {
  a[i] = 3.14;
  b[i] = 6.67;
}

#pragma omp parallel
  {
  #pragma omp for
    for(i = 0; i<N; i++)
      dot_prod = dot_prod + a[i] * b[i]; // sum up the element-wise product of the two
                                              arrays
  }

  printf("Dot product of the two vectors is %g\n", dot_prod);

  return 0;
}
```

2. In line 23 of Code 7 in Section 7.3.3 the static scheduler was demonstrated. How would the output of this code change if the dynamic scheduler were used instead?
3. In Code 8 of Section 7.3.4 the sections pragma was introduced. What would happen to Code 8 if the number of OpenMP threads were fewer than the number of sections?
4. Write a matrix–vector multiply and parallelize with OpenMP directives.

REFERENCE

1. OpenMP, The OpenMP API Specification. [Online] http://www.openmp.org/specifications/.

CHAPTER

THE ESSENTIAL MPI

8

CHAPTER OUTLINE

8.1 Introduction **250**
8.2 Message-Passing Interface Standards **251**
8.3 Message-Passing Interface Basics **253**
8.3.1 mpi.h 253
8.3.2 MPI_Init 253
8.3.3 MPI_Finalize 254
8.3.4 Message-Passing Interface Example—Hello World 254
8.4 Communicators **255**
8.4.1 Size 256
8.4.2 Rank 256
8.4.3 Example 257
8.5 Point-to-Point Messages **258**
8.5.1 MPI Send 259
8.5.2 Message-Passing Interface Data Types 259
8.5.3 MPI Recv 259
8.5.4 Example 260
8.6 Synchronization Collectives **262**
8.6.1 Overview of Collective Calls 262
8.6.2 Barrier Synchronization 263
8.6.3 Example 264
8.7 Communication Collectives **265**
8.7.1 Collective Data Movement 265
8.7.2 Broadcast 268
8.7.3 Scatter 269
8.7.4 Gather 271
8.7.5 Allgather 272
8.7.6 Reduction Operations 274
8.7.7 Alltoall 277
8.8 Nonblocking Point-to-Point Communication **279**
8.9 User-Defined Data Types **281**
8.10 Summary and Outcomes of Chapter 8 ...... **283**
8.11 Exercises **283**
References **284**

High Performance Computing. https://doi.org/10.1016/B978-0-12-420158-3.00008-3

8.1 INTRODUCTION

A major form of high performance computing (HPC) systems that enables scalability is the distributed-memory multiprocessor. Both massively parallel processors (MPPs) and commodity clusters are examples of system-level architectures of this form. The distinguishing property of this important class of supercomputer is that the main memory of the system is partitioned into fragmented components, each associated with one or more processor cores and ancillary components that together comprise what has become casually referred to as a "node". Multiple nodes integrated by means of one or more interconnect networks constitute the full high performance computer. A distributed-memory system is such that a processor core is able to access the memory intrinsic to its resident node directly, but not the memory or the external nodes making up the total system. Exchange of data, cooperation and coordination of the tasks running on the separate nodes, and overall operation of the system as a single entity are achieved through the transfer of messages between nodes by means of the system area network tying all the pieces together. Logically this is achieved through the passing of messages between pair-wise executing processes, or sometimes among more than two processes at a time. The major advantage of the distributed-memory multiprocessor is its scalability. Within the constraints of power and cost, essentially any number of nodes can be incorporated within a single supercomputer. A somewhat more nuanced value is that the programmer is forced and therefore motivated to manage the program locality explicitly, making pieces of work fit within the confines of an individual node. This has resulted in a generation of scalable application software and libraries that has achieved a million times greater throughput performance than previous-generation computing models and architecture classes. How to use this successful class of high performance computer is the subject of this chapter, and the topic and means of doing so is the message-passing interface (MPI).

Over as many as 3 decades there have been many software application programming interfaces and implementation libraries that supported the communicating sequential processes model of computation, casually referred to as the "message-passing model". These were developed by industry, within academia, and from national labs and centers, among others. But by far the most significant has been MPI [1]. MPI was, and in its most recent versions still is, a community-driven specification. Starting in late 1992, representatives of industry, government, and academia began a community-led process to develop a standard programming interface based on principles first laid out by Anthony Hoare in the mid-1970s. The strength of this approach to community building was the ready acceptance of the result and the rapid development of useful applications. The weakness was that to achieve an agreed-upon initial standard, many more controversial semantics, constructs, and mechanisms were initially discarded for the sake of unity, resulting in a more simplistic and admittedly limited interface. However, in spite of such sacrifices of sophistication, this proved to be the right path for evolutionary progress that was much needed at the time. At the risk of hyperbole, there was probably no greater achievement of practical utility for the advancement of HPC than the development of MPI. Even in its most basic form, MPI has proven a powerful, flexible, and usable programming interface. With its hundreds of commands it deals with a rich and diverse set of circumstances, yet a very small subset of these is sufficient to write a wide array of parallel applications. This chapter presents only a small subset of the total set of possible commands, but in doing so gives the student a powerful set of tools for harnessing distributed-memory supercomputers and empowering the solution of computational end-user problems.

8.2 MESSAGE-PASSING INTERFACE STANDARDS

William D. Gropp. *Photo Courtesy NCSA*

William "Bill" Gropp is an American scientist who helped develop the MPI message-passing standard. He is also coauthor of the MPICH implementation of MPI. Apart from contributing to two very influential books, "Using MPI" and "Using MPI 2", he is also a designer of the widely used Portable, Extensible Toolkit for Scientific Computation library discussed in this textbook. Among his many honors, William Gropp received the IEEE Sidney Fernbach Award in 2008 "for outstanding contributions to the development of domain decomposition algorithms, scalable tools for the parallel numerical solution of partial differential equations (PDEs), and the dominant HPC communications interface".

From 1992 to 1994 a community representing both vendors and users decided to create a standard interface for message-passing calls in the context of distributed-memory parallel computers, principally early MPPs like the Intel Touchstone Paragon. MPI-1 was the result. From the very beginning it was "just" an application programming interface (API), not a language. This was achieved by adding constructs for parallelism, data exchange communication, synchronization, and collectives through bindings to existing conventional sequential programming languages—initially Fortran 77 and C. Language bindings permit the semantics and syntax of existing languages to be exploited from the frameworks of libraries for concurrency management. These bindings allowed the widest possible use of existing application kernels, compilers, and user skill sets while augmenting them with the needed concepts of communication frameworks for coordination, cooperation, and concurrency. The MPI standard can be found online [1].

Probably equally as important as the community-derived API was for the MPI was the first reduction to practice: the first reference implementation, called "MPI over CHameleon (MPICH)" and developed at Argonne National Laboratory [2]. This was delivered in 1995 and served as the template for the many other implementations of MPI to come afterwards. Led by William Gropp, the MPICH project provided both important experience in the implementation of MPI and a platform upon which the earliest practical applications were developed and run on the MPP systems of the time, such as the CM-5 (see Fig. 8.1).

Many lessons were learned about correctness, performance, portability, and user productivity. Another value of the MPI standard was that it provided a strong unifying formalism throughout the HPC community but permitted distinguishing opportunities for individual vendors like Cray, IBM, and Hewlett-Packard. Vendors were able to keep their own internals and optimizations behind the interface.

Since then MPI has matured and evolved. MPI-1.1 fixed bugs that were revealed in early experience and clarified issues where subtleties and ambiguities of semantics were exposed. This continued again through MPI-1.2 and new rewrites of MPICH to improve its efficiency and scalability vastly. MPI-2 was a new standard that significantly extended the utility and richness of MPI, including new datatype constructors, one-sided communication, a strong input/output (I/O) package, and dynamic processes. Additional bindings beyond the original ones were developed, including to Fortran 90 and to C++. Since then the MPI semantics have been yet again extended, in some cases extensively for the release of MPI-3, with MPI-4 actively under development in 2017. While tremendous advances have been made since the early formulation of MPI-1, the constructs comprising the foundations of MPI still

FIGURE 8.1

A connection machine 5 (CM-5) with 512 nodes and a theoretical maximum capability of 65.5 GFlops, operational between 1991 and 1997.

Photo by Austin Mills via Wikimedia Commons provide a base-level set of interrelated concepts and constructs upon which to establish the means for parallel programming. It is from this initial starting point that this first tutorial in MPI programming is offered.

8.3 MESSAGE-PASSING INTERFACE BASICS

While the latest versions of MPI include literally hundreds of commands, a simple parallel program can be created using only three basic commands. This section describes how to do that. The bindings all assume the use of the C programming language, with which all examples and descriptions are presented.

8.3.1 MPI.H

Every MPI program must contain the preprocessor directive:

```
#include <mpi.h>
```

The mpi.h file contains the definitions and declarations necessary for compiling an MPI program. mpi.h is usually found in the "include" directory of most MPI installations. This directive can be positioned in any order with other directives, but must precede the beginning of the program with the main() call.

8.3.2 MPI_INIT

The part of the user application code that will contain function calls for MPI program constructs must begin with the single call to MPI_Init and expects arguments of the following form, returning an integer error value:

```
int MPI_Init(int *argc,char ***argv)
```

MPI_Init initializes the execution environment for MPI. This command has to be called before any other MPI call is made, and it is an error to call it more than a single time within the program. The number of arguments passed internally to all the parallel processes is pointed to by argc. The vector of the arguments' list is pointed to by argv, as is consistent with the C language and command-line argument variables passing. Every process launched by MPI_Init inherits copies of these two program argument variables and is achieved by the call:

```
MPI_Init(&argc,&argv);
```

prior to any of the other MPI calls within the application.

8.3.3 MPI_FINALIZE

In a sense, the other bookend to MPI_Init is the MPI_Finalize command. MPI_Finalize cleans up all the extraneous mess that was first put into place by MPI_Init. It brings to an end the computing environment for MPI. There are no arguments to this MPI service call, which has the following simple syntax:

```
MPI_Finalize();
```

This does not have to be the end of the entire program. Many other C statements can follow it. Also, its exact position in the code sequence is not particularly important as long as it comes after any other MPI commands in the program.

8.3.4 MESSAGE-PASSING INTERFACE EXAMPLE—HELLO WORLD

Somewhat sadly, there is a rite of passage that every neophyte programmer in just about any programming language has to go through: writing "Hello, World", a most trivial program first sketched out by Kernighan and Ritchie in their original book on C [3]. This is the most minimalist program one can imagine that actually works. Getting this far is a major milestone for a student, crossing the line from never having successfully written an actual computer program in the language of choice to being a programmer (sort of). So for the sake of tradition and with a justified nod to those giants who preceded us, here is "Hello, World" in MPI with C bindings.

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

int main(int argc,char **argv)
{
  MPI_Init(&argc,&argv);
  printf(" Hello, World!\n");
  MPI_Finalize();
  return 0;
}
```

Code 8.1. A trivial example of "Hello, World" using MPI.

The example in Code 8.1 is compiled and run using the MPICH implementation of MPI on a Beowulf-class cluster, as follows.

```
> mpicc code1.c -o code1
> mpirun -np 4 ./code1
  Hello, World!
  Hello, World!
  Hello, World!
  Hello, World!
```

The *mpicc* compiler wrapper links in the appropriate MPI libraries and gives the path to the file location of the mpi.h header. *mpirun −np 4* launches four instances of the code executable in the runtime environment. While using *mpicc* and *mpirun* to compile and launch MPI applications is very common, they are not part of the MPI standard and the specific compile and launching approach may differ for different machines. For example, on a Cray XE6 MPP, Code 8.1 is compiled and launched as follows:

```
> cc code1.c -o code1
> aprun -n 4 ./code1
  Hello, World!
  Hello, World!
  Hello, World!
  Hello, World!
```

In this MPP case, the *cc* compiler wrapper links in the appropriate MPI libraries and finds the appropriate headers while the launch script *aprun* launches the four instances of the executable in the runtime environment.

The only work performed by Code 8.1, of course, is to print the character stream "Hello, World!" on the standard I/O device, which is the user's terminal screen. But unlike the equivalent sequential version of this simple program, this string will be printed multiple times; in fact, it will print out as many times as there are processes running under MPI at the same time. Although all output lines look the same (note that the \n character in line 7 of Code 8.1 causes a new line), the actual order in which they are output is unspecified. A later example is more revealing of this nondeterminacy. The resulting parallelism is a consequence of the pairing of the MPI_Init and MPI_Finalize calls. There is no interaction among the separate processes in this example, however. To get the different processes to interact, the concept of communicators is needed.

8.4 COMMUNICATORS

The "Hello, World" example in Code 8.1 is very simple. But it represents a broad range of parallel computing known as "throughput" computing, where every hardware node is running the same program but on different local data. This can be scaled to a very large degree, and additional examples are demonstrated in succeeding chapters. But the principal weakness of this limited form of processing is that the processes on different nodes run entirely independent of each other. It is a "share nothing" modality in which the outcome of any one of the concurrent processes can in no way be influenced by the intermediate results of any of the other processes. Without interprocess interaction, this type of computing only supports pure weak scaling or capacity computing, as described earlier. It cannot enable capability or coordinated computing, both of which are far richer in parallel computational forms and functions. Key to this advance is the means by which the concurrent processes can interact. And this is achieved through the concept and implementation of "communicators".

MPI programs are made up of concurrent processes executing at the same time that in almost all cases are also communicating with each other. To do this, an object called the "communicator" is provided by MPI. A communicator has its own address space and various properties. In particular, it encompasses a set of MPI processes as well as specific attributes. It is through the communicator that the processes of which MPI consists can communicate with other processes. A communicator consists of multiple coexisting MPI processes, so a process may be associated with more than one communicator at the same time. Thus the user may specify any number of communicators within an MPI program, each with its own set of processes. However, all versions of MPI provide one common communicator, "MPI_COMM_WORLD". This communicator contains all the concurrent processes making up an MPI program and does not have to be explicitly created by the programmer. For simplicity and ease of understanding, almost all examples presented in this book take advantage of MPI_COMM_WORLD, as it manages the communications between concurrent processes.

8.4.1 **SIZE**

A communicator embodies a number of attributes, many of which may be referenced by the user program. Among those most widely used is "size". This property, as its name implies, indicates some aspect of a communicator's scale, specifically related to processes. The size of a communicator is the number of processes that makes up the particular communicator. The following function call provides the value of the number of processes of the specified communicator:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

The function name is "MPI_Comm_size", required to return the number of processes; *comm* is the argument provided to designate the communicator, recognizing that any process may be part of more than one communicator. The resulting value is returned to *size* within the process context. A typical statement for this purpose could be:

```
int size;
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

This will put the total number of processes in the MPI_COMM_WORLD communicator in the variable size of the process data context. As this is the same for all processes of the communicator, their respective copies of the variable size will receive the same value.

8.4.2 **RANK**

A second widely used attribute of a communicator is identification of each of the processes within the communicator. Every process within the communicator has a unique ID referred to as its "rank". The

MPI system automatically and arbitrarily assigns a unique positive integer value, starting with 0, to all the processes within the communicator. The MPI command to determine the process rank is:

```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

The function call "MPI_Comm_rank" indicates that the rank value of the calling process is to be returned to the process. The first argument, "comm", indicates the communicator to which the process belongs within which it requires its rank. The second argument, "rank", is the variable that will assume the value returned by the command. A typical statement for this purpose could be:

```
int rank;
MPI_Comm_size(MPI_COMM_WORLD,&rank);
```

In the case of the MPI_COMM_WORLD communicator, all the processes of the application will have a unique value of rank returned. Each process within this communicator when calling this function will receive a different value in its copy of the variable rank.

8.4.3 EXAMPLE

As a trivial case that nonetheless demonstrates the functionality of communicators and these simple but powerful commands, the following example is offered. This is a minor elaboration of the earlier and iconic "Hello, World" problem.

The purpose of this application program is for every process that exists within the MPI_COMM_WORLD communicator to identify itself by printing a statement to the standard output. The structure of this parallel program is the same as the previous, with the potentially interprocess communicating part of the code delimited by the pair of MPI_Init and MPI_Finalize commands. Between these two statements are the working parts of the program, such as the printf construct shown before. But added here are also the two service calls associated with the communicator: MPI_Comm_rank and MPI_Comm_size. The complete MPI code is given in Code 8.2.

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

int main(int argc,char **argv)
{
  int rank, size;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  printf(" Hello from rank %d out of %d processes in MPI_COMM_WORLD\n",rank,size);
  MPI_Finalize();
  return 0;
}
```

Code 8.2. Example where each process prints its rank and the MPI_COMM_WORLD communicator size.

This code is compiled and executed on a Beowulf-class cluster as follows:

```
> mpicc code2.c -o code2
> mpirun -np 4 ./code2
  Hello from rank 0 out of 4 processes in MPI_COMM_WORLD
  Hello from rank 2 out of 4 processes in MPI_COMM_WORLD
  Hello from rank 3 out of 4 processes in MPI_COMM_WORLD
  Hello from rank 1 out of 4 processes in MPI_COMM_WORLD
```

Code 8.2 illustrates the use of the two most common calls related to communicators. The two commands bracket by MPI_Init and MPI_Finalize are the MPI_Comm_rank (line 8), which determines the ID of the process, and the MPI_Comm_size (line 9), which finds the number of processes. In both cases they refer to the MPI_COMM_WORLD communicator as specified as the first operand in each of the two calling sequences. The second argument in each case indicates the process variable in which the related integer value is put. The printf I/O service call not only outputs the string "Hello" but also prints out two integers, one for the process rank which is unique for each process and the other giving the size of the MPI_COMM_WORLD communicator in terms of the number of processes it contains. The size for all processes is the same. The order of printing the output is undetermined, as mentioned before. With each process uniquely identified within the communicator, it is now possible to begin sending messages between processes.

8.5 POINT-TO-POINT MESSAGES

Among its most important functionalities, MPI manages the exchange of data between processes within a selected communicator. The medium of this exchange is referred to as messages. Messages provide point-to-point communication from a source process to a corresponding destination process, each with its own unique rank by which it is identified. In its simplest form, two commands are required to achieve the passing of a message. The sending of the message from the source process is accomplished by a send command. The receiving of the message by the corresponding destination process is accomplished by a recv command. Messages are matched between the two commands. While there are a number of variants of both the send family and the recv family of commands, the most basic of these are MPI_Send and MPI_Recv.

The message specification can be considered as a combination of the connection and the data of the message. The connection describes the points forming the communication. These include the following.

1. The source process rank.
2. The destination process rank.
3. The communicator of which both processes are a part.
4. The tag, which is a user-controlled value that can be used to discriminate among a set of possible messages between the same two processes.

8.5.1 MPI SEND

The send function is used by the source process to define the data and establish the connection of the message. The send construct has the following syntax:

```
int MPI_Send (void *message, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

There are six arguments to the MPI_Send call to provide this information. The first three operands establish the data to be transferred between the source and destination processes. The first argument points to the message content itself, which may be a simple scalar or a group of data. The message data content is described by the next two arguments. The second operand specifies the number of data elements of which the message is composed. These are all the same in form. The third operand indicates the data type of the elements that make up the message (see next subsection). These three values give the data to be moved by the message. The connection of the message is established by the second three operands: the rank of the destination process, the user-defined tag field, and the communicator in which the source and destination processes reside and for which their respective ranks are defined.

8.5.2 MESSAGE-PASSING INTERFACE DATA TYPES

MPI defines its own data types. This might appear redundant, as programming languages like C explicitly define data types as well. But for the sake of robustness where different processes may be written in different languages or run on different kinds of processor architectures, MPI makes explicit what is intended. Like other interfaces, MPI provides a set of primitive data types. More complex structured data types can be user defined, as is shown in a later subsection. The most common primitive data types are presented in Table 8.1, along with the C data type equivalents.

8.5.3 MPI RECV

The MPI_Recv command mirrors the MPI_Recv command to establish a connection between the source and destination processes within the specified communicator. Like the send command (MPI_Send), the receive command (MPI_Recv) describes both the data to be transferred and the connection to be established. The MPI_Recv construct is structured as follows:

```
int MPI_Recv (void *message, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

The information provided to describe the data to be exchanged is represented in a form similar to the operands of the MPI_Send command. The message itself is placed in a buffer variable, designated

Table 8.1 Some of the Basic MPI Data Types and Their C Data Type Equivalent

MPI Data Type	C Data Type Equivalent
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	No direct equivalent but like unsigned char; just 1 byte

here as "message". The number of data elements making up the full message is given by the integer count. The data type of the element of the message is one of the MPI data types defined in the previous subsection or a user-defined data type (described later).

The connection information of the MPI_Recv command is similar but not identical to that of the MPI_Send command. The source field designates the rank of the process sending the message. As before, a tag variable is given for a user-defined integer that is provided in the Send command and can be extracted for user code manipulation by the receiving process. As in all cases, the communicator in which both processes reside is specified. A final argument variable, "status", is included as the final operand of MPI_Recv. This is a record of two fields about the actual message received: the first indicates the process rank from which the message was actually received, and the second field provides the tag.

8.5.4 EXAMPLE

Code 8.3 presents a third example based on "Hello, World" to illustrate MPI commands, in this case the send and receive commands. This example expands our experience in three important ways.

1. It shows the syntactical details for setting up the information, including declarations for the MPI commands to be used.
2. It illustrates an important idiom related to how to control concurrent execution and the idea of the manager—worker form of computing using MPI.
3. It solves the problem of the previous examples that we have seen with the nonsequential printf commands.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <string.h>

int main(int argc,char **argv)
{
  int rank, size;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int message[2];  // buffer for sending and receiving messages
  int dest, src;  // destination and source process variables
  int tag = 0;
  MPI_Status status;

  // This example has to be run on more than one process
  if ( size == 1 ) {
    printf(" This example requires more than one process to execute\n");
    MPI_Finalize();
    exit(0);
  }

  if ( rank != 0 ) {
   // If not rank 0, send message to rank 0
    message[0] = rank;
    message[1] = size;
    dest = 0; // send all messages to rank 0
    MPI_Send(message, 2,MPI_INT,dest,tag,MPI_COMM_WORLD);
  } else {
   // If rank 0, receive messages from everybody else
    for (src=1;src<size;src++) {
      MPI_Recv(message,2,MPI_INT,src,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
      // this prints the message just received. Notice it will print in rank
      // order since the loop is in rank order.
      printf("Hello from process %d of %d\n",message[0],message[1]);
    }
  }

  MPI_Finalize();
  return 0;
}
```

Code 8.3. "Hello" example where all processes with ranks greater than 0 send their rank and size to the process with rank 0 for printing.

```
> mpicc code3.c -o code3
> mpirun -np 4 ./code3
  Hello from process 1 of 4
  Hello from process 2 of 4
  Hello from process 3 of 4
```

Much of this example is similar to the previous ones shown in this chapter. Commands such as MPI_Init, MPI_Finalize, MPI_Comm_rank, and MPI_Comm_size are all the same in their usage. And as in the other examples, the communicator used is MPI_COMM_WORLD. But at this point the similarities end.

The biggest difference is the important idiom of manager—worker organization, in which one process, the manager, coordinates the execution of the other processes, the workers. Sometimes the manager is referred to as the "root" process. All processes, whether root or worker, receive and execute the same process code (procedure). Thus it is within the user code itself that the distinction between manager and worker has to be prescribed. In this example, the manager is assumed to be of rank = 0 and the workers are identified as $1 < \text{rank} < \text{size} - 1$. Hence the code is separated between manager and workers by the conditional on line 24. If true, a message array of size 2 is populated with the rank and size variables. The message is then sent using the MPI_Send command to the destination process (line 28), which is always rank 0 in this case.

The magic occurs in the body of code executed by the root process within the otherwise bounded sequence in line 30. The ordered iterative loop embodied by the for block (line 32) accepts messages using the MPI_Send command in rank-ordered fashion and prints them out in that order, guaranteeing the sequence of outputs. The control by the root process makes certain that the output information from the worker processes is presented in a deterministic form, i.e., a rank-ordered list. This is an important idiom of control in MPI using the manager—worker paradigm. Because only one message is sent from each nonroot process, the MPI_Recv command is told to ignore the tag with the useful MPI_ ANY_TAG field (line 33).

8.6 SYNCHRONIZATION COLLECTIVES

While point-to-point communication is the backbone of MPI management of data exchange, additional communication constructs that involve more processes at one time are a powerful addition to simplifying MPI programming and improving performance efficiency. These are referred to as "collective operations" or simply "collectives".

8.6.1 OVERVIEW OF COLLECTIVE CALLS

A communication pattern that encompasses all processes within a communicator is known as "collective communication". One of the important aspects of a communicator is the set of processes within an MPI program to which the programmer wants to apply collective operators, and this may not be all the processes used by the program as a whole. MPI has several collective communication calls. The most frequently used are synchronization collectives, communication collectives, and reduction collective operators. Synchronization collective operations bring all the processes of a communicator up to a known place in the control flow even though their separate processes are executing asynchronously, some further ahead than others. Communication collectives exchange data in different patterns among more than two (point-to-point) processes within a communicator. Reduction collective operators act as a common communicative operator across versions of the same variable of all the processes. The next subsection briefly describes the simplest of synchronization collectives, the global barrier.

8.6.2 BARRIER SYNCHRONIZATION

The MPI_Barrier command creates, as the name implies, a point of barrier synchronization among all the processes of the specified communicator. This command has a simple syntax of a single operand:

```
int MPI_Barrier (MPI_Comm communicator)
```

The `communicator` is the communicator of the processes engaged in the synchronization. The barrier requires that all the processes reach that point in their respective code, and then wait for all the other processes of the communicator to do the same before proceeding with their separate computations. Thus all processes block at the point of the barrier until they determine that all other processes are there as well.

Fig. 8.2 illustrates the barrier operation.

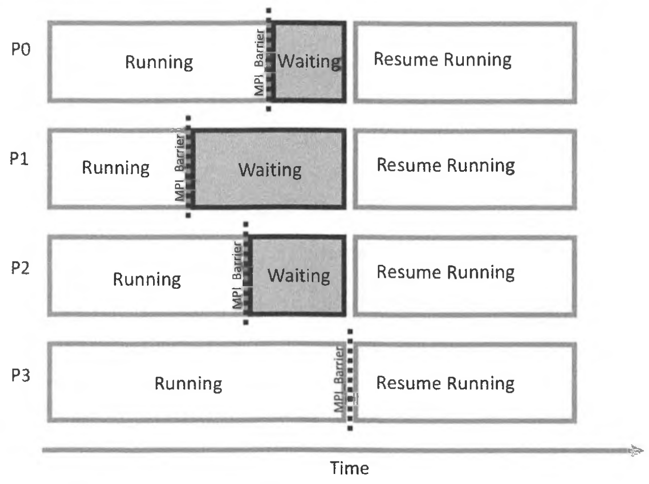


FIGURE 8.2

An illustration of the MPI_Barrier operation. Processes P0 through P3 enter the point of barrier synchronization at different times and potentially in unpredictable order. None of the processes proceeds beyond this point in the computation until all the processes reach this point. Only then do the four processes continue on to their next operations. In this way, all processes can be assured that the others have completed the necessary work. This can be an important condition to avoiding a number of different failure modes resulting from the uncertainty imposed by asynchronous operation.

8.6.3 EXAMPLE

A somewhat artificial example of the use of the MPI_Barrier collective command is presented below to demonstrate it syntax. This is an extension of the "Hello, World" example. It is also an opportunity to introduce another occasionally useful MPI instruction, MPI_Get_processor_name, which gives access to the actual hardware for purposes of identification. Depending on the MPI implementation, this might simply be the output from *gethostname* or may be something more detailed.

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

int main(int argc,char **argv)
{
  int rank, size,len;
  MPI_Init(&argc,&argv);
  char name[MPI_MAX_PROCESSOR_NAME];

  MPI_Barrier(MPI_COMM_WORLD);

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Get_processor_name(name,&len);

  MPI_Barrier(MPI_COMM_WORLD);

  printf(" Hello, world! Process %d of %d on %s\n",rank,size,name);

  MPI_Finalize();
  return 0;
}
```

Code 8.4. Example of MPI_Barrier and MPI_Get_processor_name.

```
> mpicc code4.c -o code4
> mpirun -np 4 ./code4
  Hello, world! Process 2 of 4 on cutter01
  Hello, world! Process 3 of 4 on cutter01
  Hello, world! Process 0 of 4 on cutter01
  Hello, world! Process 1 of 4 on cutter01
```

The example code above inserts two synchronization points with the two highlighted instances of the MPI_Barrier command. The first is just before the conventional MPI commands getting the size of the MPI_COMM_WORLD communicator and the unique rank identifiers of the individual process within that communicator. The second barrier is just after the newly introduced MPI_Get_processor_name command. Every process is blocked at both points until all processes have arrived at the respective barrier.

The MPI_Get_processor_name reminds the student that there is a difference between the abstraction of the executing process and the physical processor core resource upon which the process is computing. This command, as the name implies, acquires the character string that MPI uses to represent each processor core uniquely. In the example above, this character string is simply the output from *gethostname* which was *cutter01*, the name of the compute node on which the example was run. MPI has a lot of special constants that describe key values of its operation. Here one is used to specify the greatest possible length of the character string representing the processor name. This constant is MPI_MAX_PROCESSOR_NAME, and is referred to near the beginning of the code where the variable "name" is declared as a character buffer.

8.7 COMMUNICATION COLLECTIVES

Communication collective operations can dramatically expand interprocess communication from point-to-point to n-way or all-way data exchanges. These commands can greatly simplify user programming and provide the opportunity for greater execution efficiency by telling MPI what one actually wants to happen. Communication collective operations are among the most powerful contributing capabilities of MPI for weaving many individual processes into a single scalable computation. While there are many variants of communication collectives, a few are very widely employed in support of parallel algorithms and are described in this section.

8.7.1 COLLECTIVE DATA MOVEMENT

Collective data movement relates to different patterns by which compound data may be exchanged among concurrent processes within a specific communicator. The requirements for these data distributions are a function of the parallel algorithms being employed and the degree to which intermediate results of any process need to be shared with one or more other processes to continue the evolving distributed computation. Such patterns can be diverse, but four basic patterns satisfy most algorithmic requirements of data exchange: broadcast, scatter, gather, and allgather. These are illustrated in Figs. 8.3–8.6.

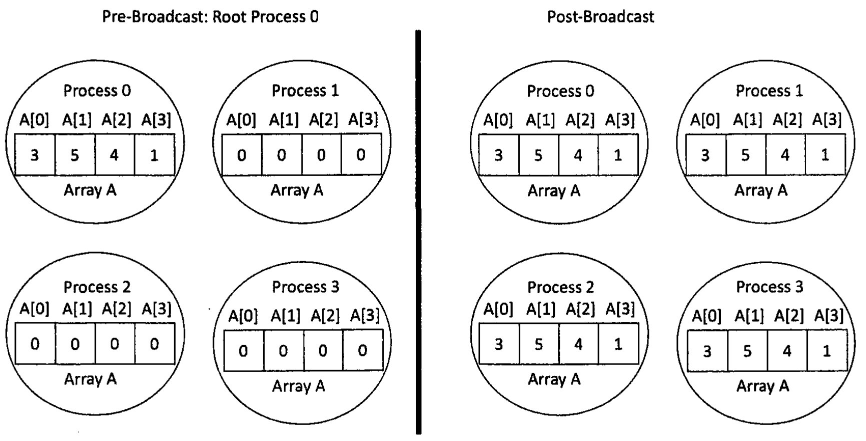


FIGURE 8.3

The broadcast operation. Broadcast shares a value or structure that exists within the context of one process with all the other processes of a communicator. In this example the root process, 0, shares the integer array A of length 4 with all the other processes.

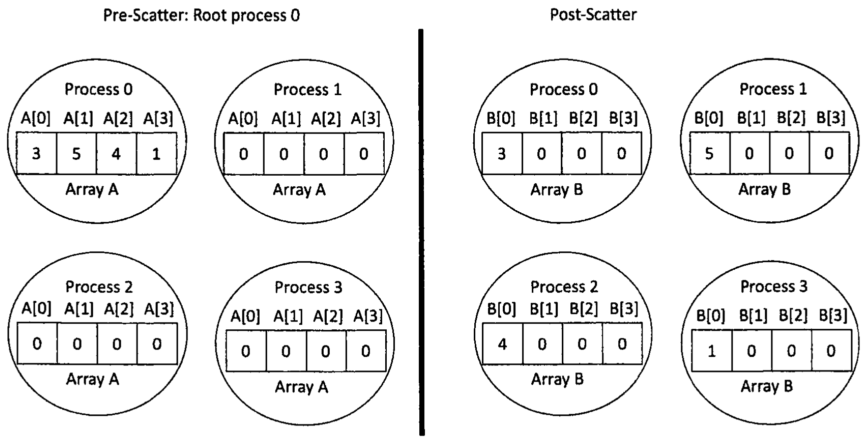


FIGURE 8.4

The scatter operation. The scatter collective communication pattern, like broadcast, shares data of one process (the root) with all the other processes of a communicator. But in this case it partitions a set of data of the root process into subsets and sends one subset to each of the processes. To be clear, each receiving process gets a different subset, and there are as many subsets as there are processes. In this example the send array is A and the receive array is B. B is initialized to 0. The root process (process 0 here) partitions the data into subsets of length 1 and sends each subset to a separate process.

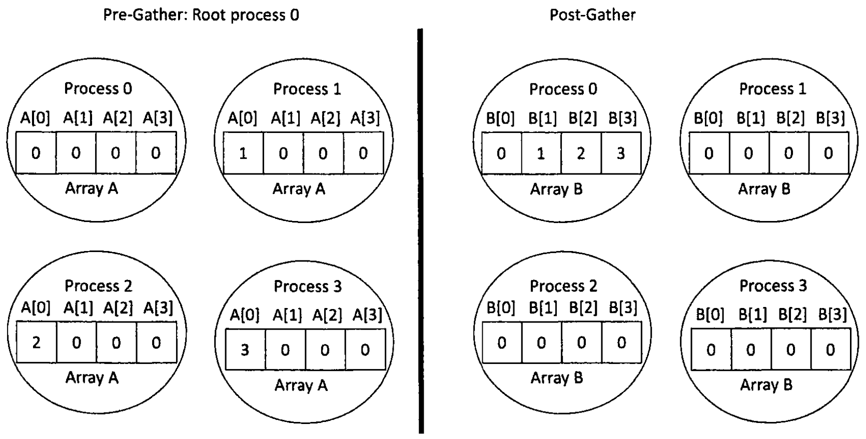


FIGURE 8.5

The gather operation. The gather collective communication pattern is, in a sense, the opposite of the scatter collective. In the case of the gather, as the name suggests, data from all the processes is sent to the root process, which is gathering up the data from the other processes. Of course, it is actually each process sending its respective designated data to the consumer process which organizes all the separate data partitions into one cumulative structure. In this example A is the send array and B is the receive array. B is initialized to 0 prior to the gather.

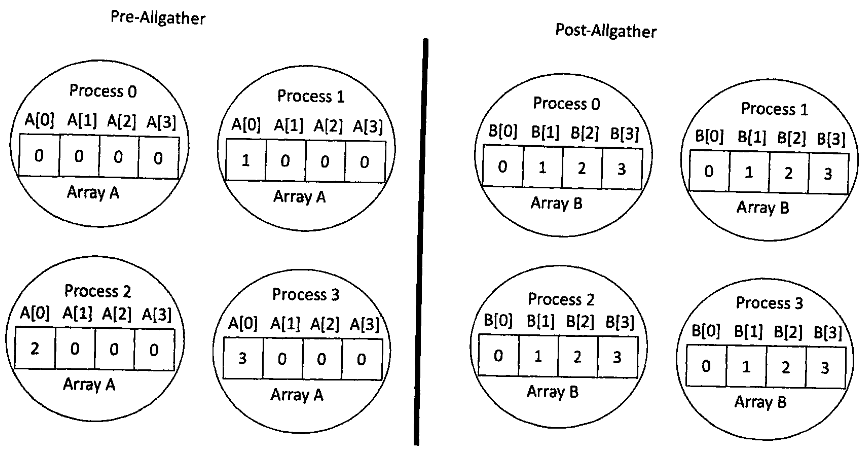


FIGURE 8.6

The allgather operation. This operation is equivalent to a gather operation followed by a broadcast of the array so that each process contains an identical receive array. In this example the A array is the send array and the B array is the receive array. B is initialized to 0.

Broadcast, illustrated in Fig. 8.3, shares a value or structure that exists within the context of one process with all the other processes of a communicator. As shown in the first diagram, the values in the A integer array in process 0 are copied to the equivalent arrays in all the other processes so they all have the same information. Broadcast, like other collective communications, provides the means by which the intermediate results of any one process are efficiently shared with all the other processes.

The scatter collective communication pattern, illustrated in Fig. 8.4, like broadcast shares data of one process with all the other processes of a communicator. But in this case it partitions a set of data of one process into subsets, and sends one subset to each of the processes. Each receiving process gets a different subset and there are as many subsets as there are processes. The process 0 has a set of data that is partitioned, in this case into four distinct partitions, A[0], A[1], A[2], and A[3], which is equal to the number of processes, processes 0 through 3 of the communicator. The first partition is returned to the source process, process 0. Data partition A[1] is sent to the second process, process 1. Partition A[2] is sent to process 2, and so on. In this way the original data in process 0 is distributed equally among all the processes of the communicator.

The gather collective communication pattern, illustrated in Fig. 8.5 is in a sense the opposite of the scatter collective. In the case of the gather, data from all the processes is sent to a particular process which is gathering up the data from the other processes. Of course, it is actually each process sending its respective designated data to the consumer process which organizes all the separate data partitions into one cumulative structure.

The extension of gather that makes it possible for all processes to use the results across the entire communicator is the allgather illustrated in Fig. 8.6. This is equivalent to first performing a gather of data from all the processes to a single receiving process and then broadcasting the accumulated data back to all the processes so that all processes have all of the resulting data.

8.7.2 BROADCAST

The broadcast communication collective operation is perhaps the simplest of the collectives, and among the most important as well. As described above, it permits a message incorporating data from a source process to be shared with all processes of a communicator. The syntax of the broadcast operation takes the following form:

```
Int MPI_Bcast (void *shared_data, int number, MPI_Datatype datatype,
int source_process, MPI_Comm communicator)
```

The broadcast operation is achieved through the MPI_Bcast command in MPI. The operands define the form and source of the data to be sent to all the processes. The broadcast is performed within the scope of a communicator specified by the last argument, or the "communicator" of type MPI_Comm, and the broadcast data are sent to all the processes within it. The data come from a single process identified by its rank within the communicator by source_process (or root process), which is the penultimate argument of MPI_Bcast. Like many other message-passing commands, the data to be sent is determined by the first three arguments: the name of the variable pointing to the data buffer, here "shared_data" in the first argument, the type of data elements of which it is composed, here "datatype" of type MPI_Datatype in the third argument, and the "number" of elements of data type making up the data to be broadcast.

The equivalent MPI code to the broadcast illustrated in Fig. 8.3 is given in Code 8.5.

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

int main(int argc,char **argv)
{
  int rank, size,i;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  int A[4];

  // Initialize array
  for (i=0;i<4;i++) {
    A[i] = 0;
  }

  int root = 0; // Define a root process

  if (rank == root ) {
    // Initialize array A
    A[0] = 3;
    A[1] = 5;
    A[2] = 4;
    A[3] = 1;
  }

MPI_Bcast(A,4,MPI_INT,root,MPI_COMM_WORLD);

printf(" Rank %d A[0] = %d A[1] = %d A[2] = %d A[3] = %d\n",
         rank,A[0],A[1],A[2],A[3]);

 MPI_Finalize();
 return 0;
}
```

Code 8.5. An example of MPI_Bcast that corresponds to the illustration in Fig. 8.3.

```
> mpirun -np 4 ./code5
Rank 0 A[0] = 3 A[1] = 5 A[2] = 4 A[3] = 1
Rank 2 A[0] = 3 A[1] = 5 A[2] = 4 A[3] = 1
Rank 1 A[0] = 3 A[1] = 5 A[2] = 4 A[3] = 1
Rank 3 A[0] = 3 A[1] = 5 A[2] = 4 A[3] = 1
```

8.7.3 SCATTER

The communication collective operation "scatter" distributes data of one process in separate parts to all the processes (including itself) within the scope of a communicator. The communicator of size processes disseminates the data of the source process in size-equal partitions. The distribution is in rank order across the set of processes and the linear dimension of the dataset. This is a particularly important construct for scalable matrices across a distributed-memory system.

The scatter operation is performed by means of the MPI_Scatter command. The operands define the form and source of the data to be sent. No destination identifier is required, as all processes are implicitly included as receiving part of the distributed data. The syntax of the scatter operation takes the following form:

```
int MPI_Scatter (void *send_data, int send_number, MPI_Datatype datatype,
void *put_data, int put_number, int source_rank, MPI_Comm communicator)
```

The scatter is performed within the scope of a communicator specified by the last argument, here "communicator" of type MPI_Comm, and the data is sent to all the processes within it. The data comes from a single process identified by its rank within the communicator by source_rank which is the penultimate argument of MPI_Scatter(). Like many other message-passing commands, the data to be sent is determined by the first three arguments: the name of the data, here "shared_data" in the first argument, the type of data elements of which it is composed, here "datatype" of type MPI_Datatype in the third argument, and the "send_number" of elements of data type making up the data to be distributed. Where the data is to be put at the receive processes is specified by put_data and the size of the data of the data type is given by the integer put_number. The equivalent MPI code to the scatter operation illustrated in Fig. 8.4 is given in Code 8.6.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc,char **argv)
{
  int rank, size,i;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  if ( size != 4 ) {
  printf(" Example is designed for 4 processes\n");
  MPI_Finalize();
  exit(0);
 }

  // A is the sendbuffer and B is the receive buffer
  int A[4],B[4];

  // Initialize array
  for (i=0;i<4;i++) {
    A[i] = 0;
    B[i] = 0;
 }

  int root = 0; // Define a root process

  if (rank == root ) {
   // Initialize array A
    A[0] = 3;
    A[1] = 5;
    A[2] = 4;
    A[3] = 1;
 }

 MPI_Scatter(A,1,MPI_INT,B,1,MPI_INT,root,MPI_COMM_WORLD);

 printf(" Rank %d B[0] = %d B[1] = %d B[2] = %d B[3] = %d\n",
            rank,B[0],B[1],B[2],B[3]);

 MPI_Finalize();
  return 0;
}
```

Code 8.6. An example of MPI_Scatter that corresponds to the illustration in Fig. 8.4.

```
> mpirun -np 4 ./code6
 Rank 0 B[0] = 3 B[1] = 0 B[2] = 0 B[3] = 0
 Rank 2 B[0] = 4 B[1] = 0 B[2] = 0 B[3] = 0
 Rank 1 B[0] = 5 B[1] = 0 B[2] = 0 B[3] = 0
 Rank 3 B[0] = 1 B[1] = 0 B[2] = 0 B[3] = 0
```

8.7.4 GATHER

The communication collective operation "gather" is in some senses the opposite of the scatter operation described above. In this case every process of a given communicator sends its respective designated dataset to the same specified process. The syntax of the gather operation takes the following form:

```
int MPI_Gather (void *send_data, int send_number, MPI_Datatype send_datatype,
 void *put_data, int put_number, MPI_Datatype put_datatype, int destination_rank,
 MPI_Comm communicator)
```

The gather operation is done through the MPI_Gather command in MPI. The operands define the form and source of the data to be sent to the single receiving process and the form and destination of the data being received. The gather is performed within the scope of a communicator specified by the last argument, here "communicator" of type MPI_Comm, and the data is sent from all the processes within it to the single receiving process. The data comes from every process within the communicator. As before, the data to be sent is determined by the first three arguments: the name of the data, here "send_data" in the first argument, the type of the data elements of which it is composed, here "send_datatype" of type MPI_Datatype in the third argument, and the "send_number" of elements of data type making up the data to be distributed. Where the data is to be put at the receive process is specified by "put_data" of type "put_datatype", and the size of the data of data type is given by the integer "put_number". The process to which all the data across the processes is accumulated in the communicator is specified by the integer argument "destination_rank", which is the seventh operand. The equivalent MPI code to the gather operation illustrated in Fig. 8.5 is shown in Code 8.7.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc,char **argv)
{
 int rank, size,i;
 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 MPI_Comm_size(MPI_COMM_WORLD,&size);

if ( size != 4 ) {
  printf(" Example is designed for 4 processes\n");
  MPI_Finalize();
  exit(0);
}

// A is the sendbuffer and B is the receive buffer
int A[4],B[4];

// Initialize array
for (i=0;i<4;i++) {
  A[i] = 0;
  B[i] = 0;
}
A[0] = rank;

int root = 0; // Define a root process

MPI_Gather(A,1,MPI_INT,B,1,MPI_INT,root,MPI_COMM_WORLD);

printf(" Rank %d B[0] = %d B[1] = %d B[2] = %d B[3] = %d\n",
         rank,B[0],B[1],B[2],B[3]);

MPI_Finalize();
return 0;
}
```

Code 8.7. An example of MPI_Gather that corresponds to the illustration in Fig. 8.5.

```
> mpirun -np 4 ./code7
 Rank 1 B[0] = 0 B[1] = 0 B[2] = 0 B[3] = 0
 Rank 2 B[0] = 0 B[1] = 0 B[2] = 0 B[3] = 0
 Rank 3 B[0] = 0 B[1] = 0 B[2] = 0 B[3] = 0
 Rank 0 B[0] = 0 B[1] = 1 B[2] = 2 B[3] = 3
```

8.7.5 ALLGATHER

The syntax of the MPI Allgather operation is nearly identical to that of the MPI gather operation, except that there is no longer any need to provide a destination rank because of the broadcast implicit in the allgather operation.

```
int MPI_Allgather (void *send_data, int send_number, MPI_Datatype send_datatype,
 void *put_data, int put_number, MPI_Datatype put_datatype, MPI_Comm communicator)
```

The equivalent MPI code to the allgather operation illustrated in Fig. 8.6 is shown in Code 8.8.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc,char **argv)
{
  int rank, size,i;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  if ( size != 4 ) {
    printf(" Example is designed for 4 processes\n");
    MPI_Finalize();
    exit(0);
  }

  // A is the sendbuffer and B is the receive buffer
  int A[4],B[4];

  // Initialize array
  for (i=0;i<4;i++) {
     A[i] = 0;
     B[i] = 0;
  }
  A[0] = rank;

  int root = 0; // Define a root process

  MPI_Allgather(A,1,MPI_INT,B,1,MPI_INT,MPI_COMM_WORLD);

  printf(" Rank %d B[0] = %d B[1] = %d B[2] = %d B[3] = %d\n",
            rank,B[0],B[1],B[2],B[3]);

  MPI_Finalize();
  return 0;
}
```

Code 8.8. An example of MPI_Allgather that corresponds to the illustration in Fig. 8.6.

```
> mpirun -np 4 ./code8
 Rank 0 B[0] = 0 B[1] = 1 B[2] = 2 B[3] = 3
 Rank 1 B[0] = 0 B[1] = 1 B[2] = 2 B[3] = 3
 Rank 2 B[0] = 0 B[1] = 1 B[2] = 2 B[3] = 3
 Rank 3 B[0] = 0 B[1] = 1 B[2] = 2 B[3] = 3
```

8.7.6 REDUCTION OPERATIONS

Reduction collectives are similar to gather, but perform some sort of reducing operation on the gathered data such as calculating a sum, finding a maximum value, or performing some user-defined operation. Predefined reduction operations in MPI are given in Table 8.2.

The syntax for the reduction operation in MPI is as follows.

```
int MPI_Reduce (const void *send_data, void *put_data, int send_number,
MPI_Datatype datatype, MPI_OP operation,int destination_rank, MPI_Comm communicator)
```

The first two arguments are the data sent to the reduction operation by each process and the location at the destination rank is specified by "put_data", both of type "datatype". The size of the data sent is given by the "send_number". The reduction operation is either one of those listed in Table 8.2 or user defined. An example of MPI_Reduce in a vector dot product calculation is presented in Code 8.9.

Table 8.2 Predefined Reduction Operations in MPI and Supported Predefined MPI Data Types

Predefined Reduction Operation	MPI Name	Supported Type
Maximum	MPI_MAX	MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE
Minimum	MPI_MIN	MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE
Summation	MPI_SUM	MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE
Product	MPI_PROD	MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE
Logical AND	MPI_LAND	MPI_INT, MPI_LONG, MPI_SHORT
Bit-wise AND	MPI_BAND	MPI_INT, MPI_LONG, MPI_SHORT, MPI_BYTE
Logical OR	MPI_LOR	MPI_INT, MPI_LONG, MPI_SHORT
Bit-wise OR	MPI_BOR	MPI_INT, MPI_LONG, MPI_SHORT, MPI_BYTE
Logical XOR	MPI_LXOR	MPI_INT, MPI_LONG, MPI_SHORT
Bit-wise XOR	MPI_BXOR	MPI_INT, MPI_LONG, MPI_SHORT, MPI_BYTE
Maximum value and location	MPI_MAXLOC	Pair data types: MPI_DOUBLE_INT (a double and an int), MPI_2INT (two ints)
Minimum value and location	MPI_MINLOC	Pair datatypes: MPI_DOUBLE_INT (a double and an int), MPI_2INT (two ints)

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <mpi.h>
 4
 5 int main(int argc,char **argv) {
 6   MPI_Init(&argc,&argv);
 7   int rank,p,i, root = 0;
 8   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 9   MPI_Comm_size(MPI_COMM_WORLD,&p);
10
11   // Make the local vector size constant
12   int local_vector_size = 100;
13
14   // compute the global vector size
15   int n = p*local_vector_size;
16
17   // initialize the vectors
18   double *a, *b;
19   a = (double *) malloc(
20         local_vector_size*sizeof(double));
21   b = (double *) malloc(
22         local_vector_size*sizeof(double));
23   for (i=0;i<local_vector_size;i++) {
24     a[i] = 3.14*rank;
25     b[i] = 6.67*rank;
26   }
27
28   // compute the local dot product
29   double partial_sum = 0.0;
30   for (i=0;i<local_vector_size;i++) {
31     partial_sum += a[i]*b[i];
32   }
33
34  double sum = 0;
35  MPI_Reduce(&partial_sum,&sum,1,
36          MPI_DOUBLE,MPI_SUM,root,MPI_COMM_WORLD);
37
38  if ( rank == root ) {
39    printf("The dot product is %g\n",sum);
40  }
41
42  free(a);
43  free(b);
44  MPI_Finalize();
45  return 0;
46 }
```

Code 8.9. Example of MPI Reduce which computes the dot product of two vectors. The two vectors here, a and b, are initialized arbitrarily (lines 23–26). The local dot product is computed in lines 29–32, and then the partial sum of the dot product from each process is summed using MPI_Reduce in lines 35–36. Note that the global vector sizes change as a function of the number of processes used, while the size of the vector segments local to the process remains constant as is done in weak scaling tests.

The companion to MPI_Reduce is MPI_Allreduce, which behaves the same as MPI_Reduce except that the result of the reduction is broadcast to all processes in the communicator. As such, the syntax for usage is nearly identical except that no "destination rank" input is needed since all ranks receive the result.

```
int MPI_Allreduce (const void *send_data, void *put_data, int send_number,
MPI_Datatype datatype, MPI_OP operation, MPI_Comm communicator)
```

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

int main(int argc,char **argv) {

  MPI_Init(&argc,&argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank); // identify the rank

  int input = 0;
  if ( rank == 0 ) {
    input = 2;
  } else if ( rank == 1 ) {
    input = 7;
  } else if ( rank == 2 ) {
    input = 1;
  }
  int output;

  MPI_Allreduce(&input,&output,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);

  printf("The result is %d rank %d\n",output,rank);

  MPI_Finalize();

  return 0;
}
```

Code 8.10. An example of MPI_Allreduce. The sum of the input variable is computed and broadcast to all processes. If run on three processes or more, each process should have as output the value 10.

```
> mpirun -np 4 ./code10
The result is 10 rank 0
The result is 10 rank 1
The result is 10 rank 2
The result is 10 rank 3
```

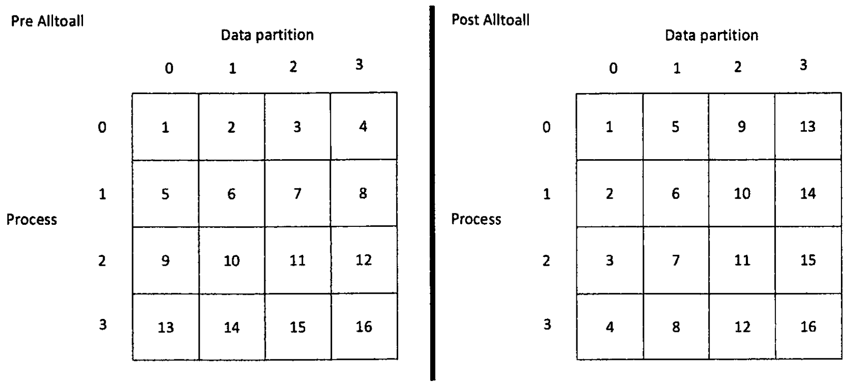
8.7.7 ALLTOALL

There is an important extension to the MPI_Allgather pattern that frequently appears in scientific computations: the alltoall communication pattern. In this pattern, distinct data is sent to each of the receivers and each sender is also a receiver. When displayed as a matrix with rows representing processes and columns representing data partitions, the alltoall communication pattern looks exactly like the matrix transpose illustrated in Fig. 8.7.

The MPI_Alltoall operation has the following syntax:

```
int MPI_Alltoall (void *send_data, int send_number, MPI_Datatype send_datatype,
 void *put_data, int put_number, MPI_Datatype put_datatype, MPI_Comm communicator)
```

As an extension to MPI_Allgather, MPI_Alltoall takes the exact same arguments as MPI_allgather although the communication pattern is different, as illustrated in Fig. 8.7. The MPI version of the operation illustrated in Fig. 8.7 is shown in Code 8.11.



Pre Alltoall

Process \ Data partition	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

Post Alltoall

Process \ Data partition	0	1	2	3
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

FIGURE 8.7

The alltoall communication pattern extends allgather, where distinct data is sent to each receiver and each sender is also a receiver. The ith data partition is sent to the jth process. The communication pattern looks like a matrix transpose when listing the data in each process in rows and the data partitions on each process as the columns. In this example, each data partition on each process only contains a single integer and the number of processes has been limited to four to see the alltoall communication pattern better.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc,char **argv) {

  MPI_Init(&argc,&argv);
  int rank,size,i;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if ( size != 4 ) {
    printf(" This example is designed for 4 proceses\n");
    MPI_Finalize();
    exit(0);
  }

  int A[4],B[4];

  for (i=0;i<4;i++) {
    A[i] = i+1 + 4*rank;
  }

  // Note that the send number and receive number are both one.
  // This reflects that fact that the send size and receive size
  // refer to the distinct data size sent to each process.
  MPI_Alltoall(A,1,MPI_INT,B,1,MPI_INT,MPI_COMM_WORLD);

  printf("Rank: %d B: %d %d %d %d\n",rank,B[0],B[1],B[2],B[3]);

  MPI_Finalize();

  return 0;
}
```

Code 8.11. The MPI example that corresponds to the illustration in Fig. 8.7.

```
> mpirun -np 4 ./code11
Rank: 0 B: 1 5 9 13
Rank: 1 B: 2 6 10 14
Rank: 2 B: 3 7 11 15
Rank: 3 B: 4 8 12 16
```