ness of near-neighbor connections, but the utility of such devices is limited to the calculations that fit the specific array geometries. Nevertheless, such devices could be put in high production and swing the cost-performance pendulum to favor near-neighbor communication. On the other hand, breakthroughs in optical transmission and optical switching may swing the balance towards the perfect shuffle. Such advances would make communication faster over the longer interconnections, and reduce the cost of sending data much further than the nearest neighbor.

Consequently, new advances in architectures for the continuum model are driven by the advances yet to come in devices and communications.

4.6 Architectures for the Continuum Model—Which Direction?

The continuum model is a natural model for parallelism. Near-neighbor interactions can be modeled by networks of processors connected together as near-neighbors. The advantage of the near-neighbor structure is very strong for those problems that are ideally matched to such a structure.

In a broad spectrum of problems, as the fit becomes less ideal, the performance of near-neighbor connections becomes poorer and poorer, to the extent that gains due to parallel execution are offset by the inefficient use of hardware. Here are the basic choices available to the architect:

1. Build a highly specialized, near-neighbor architecture that is very fast and effective for some class of problems within the continuum model.
2. Build a somewhat more general machine, but maintain high speed for the continuum model. Provide extra capability through richer interconnections, such as the perfect shuffle, and through other mechanisms that provide speed enhancement for problems that fall outside the continuum model.
3. Build a very general parallel machine that has broad applicability, including the continuum model, although its speed for continuum calculations may not be as high as for an architecture specialized for the class of problems.

The potential size of the user community increases by one to two orders of magnitude as you move from the first to the second choice, and again as you move from the second to the third choice. A large user base tends to provide cost reductions to each user because they have to support a much smaller share of the hardware and software development costs.

A large demand also provides greater profit motivation, but if a designer chooses to serve the large community and produces a fairly general architecture, the users who absolutely need a machine for the continuum model will be unsatisfied if the general architecture is significantly slower than an architecture specialized for the continuum model. Moreover, this same user group will question the value to themselves of the hardware and software that support the more general classes of problems, since this group of users may be paying for these aspects of the computer system and yet derive no discernible benefit from them.

Which community should the architect serve? There is no obvious answer to this question. The architect should be prepared to build any of the possible machines, from the most specialized to the most general, each optimized for the best possible cost and performance for that architecture.

Market forces and other priorities will dictate which machine actually gets built. Some developers will choose the most general approach, and hope to install many copies of a machine. Some developers will choose to a carve a niche for their ideas by producing a relatively small number of copies of a highly specialized machine. Yet other developers may choose a design that falls in between.

Whichever choice is made, the architecture has to be cost-effective for the user community. For the smaller markets, a significant portion of the challenge is to keep hardware and software development costs low, so that these costs when amortized over copies actually sold are still within reasonable bounds. Thus, not only must the architect produce a cost-effective design, but the design process itself must be done efficiently.

One important observation from this chapter is that what appears to be an ideal architecture for a class of problems may not be ideal at all. An architect who produces a machine that executes a particular code very efficiently may be somewhat disappointed when research advances in basic algorithms produce a new, efficient solution technique not at all suited to the specific architecture. In such a case the very specialized machine may have difficulty competing with a less specialized machine that happens to be able to run the more efficient algorithm.

Breakthroughs do occur from time to time, such as with the formulation of the fast Fourier transform [Cooley and Tukey 1965]. Prior to their work, the best algorithm required N^2 multiplications and required a particular type of access to data. The newer, faster algorithm requires only $N \log N$ multiplications and uses a very different data flow. A machine built for the older algorithm would not serve the newer one well. The more specialized the architecture, the more susceptible it is to competitive methods when breakthroughs do occur. The architect of the specialized machine has to assess the risk of a breakthrough. For the continuum model, the risks are high enough to merit attention.

In recent years, algorithm improvements have changed the basic flow of data in various solution techniques, have altered the grid structure that models the continuum, and have even provided for multiple grid spacing. A machine built specifically for algorithms of 20 years ago would do relatively poorly when executing some of the new algorithms for the same problems.

As an example of the evolution of parallel algorithms, the fast algorithms for the continuum model described earlier in this chapter may make better use of connection patterns like the perfect shuffle than of connection patterns that are near-neighbor mesh connections, but the near-neighbor connections were the backbone for the first large-scale computers for the continuum. Another step in the evolution is represented by the Cosmic Cube described earlier in this chapter, which in a sense combines the near-neighbor interconnections and the perfect shuffle. It uses near-neighbor connections in six dimensions, but at best only three of those dimensions lead to short interconnections in a three-dimensional packaging world. The other three dimensions force interconnections to have relatively long physical lengths.

The six-dimensional connection structure of the Cosmic Cube gives the same adjacency pattern achieved by the perfect shuffle. The difference is that all dimensions are adjacent at all times in a Cosmic Cube, whereas the adjacency changes in time in a perfect shuffle structure. Because processors that are directly connected within a Cosmic Cube have indices that differ by a single power of 2, this structure is well suited for recursive doubling, cyclic reduction, Fourier transforms, and other applications mentioned in this section.

Hoshino [1989], on the other hand, has shown that for the general class of scientific calculations the overwhelming majority of processor-to-processor interactions occur across near-neighbor links on a two-dimensional mesh. The additional connectivity provided by a hypercube and the greater distances spanned by the perfect shuffle rarely come into play, and provide only a marginal decrease in the number of operations while contributing greatly to cost. He provides a strong case for two-dimensional mesh connections based on extensive experience in implementing scientific applications. Even though his applications occasionally force some processors to communicate over long distances, this happens sufficiently infrequently that it degrades performance only slightly. Hence, Hoshino's case rests on the fact that the communication constraints imposed by a two-dimensional mesh do not degrade performance of actual programs. Indeed, his PAX architecture is a compromise between the ILLIAC IV and Cosmic Cube architectures, incorporating some good features of each together with some features unique to PAX.

Nevertheless, experience with parallel applications is still rather limited but growing every year. New techniques and new algorithms are still appearing in abundance. As these appear, they force us to rethink our conclusions on what combination of algorithms, architecture, applications and produces an efficient way to solve problems.

In summary, there is no obvious best design for parallel processors for the continuum model. The available approaches depend on how specialized the processing system can be. A processor for the continuum model undoubtedly will be somewhat specialized—it will probably have an interconnection system to speed up typical programs for this model. Which approach, if any, becomes dominant is most likely to depend on the directions of device technology in the coming years, with near-neighbor structures dependent on VLSI advances and perfect-shuffle structures dependent on advances in interconnections technology.

Exercises

4.1 The object of this exercise is to explore calculations for the continuum model. Assume that you have a square array of points, 9 × 9, and that the value of the potential function on the boundary is 0 on the top row, and is 10 along all other boundary points.

a) Initialize the potential function to 0 on all interior points. Calculate the Poisson solution for the values of all interior points by replacing each interior point with the average value of each of its neighboring points. Compute the new values for all interior points before updating any interior points. Run this simulation for five iterations and show the answers you obtain at this point. Then run until no interior point changes by more than 0.1 percent, and count the total number of iterations until convergence. This method is usually called the *Jacobi method*. Note: The values on the boundary are fixed and do not change during the computation.

b) Repeat the process in the previous problem, except update a point as soon as you have computed the new value and use the new value when you reach a neighboring point. You should scan the interior points row by row from top to bottom and from left to right within rows. This method is usually called the *Gauss-Seidel method*.

c) The second process seems to converge faster. Give an intuitive explanation of why this might be the case.

d) How do your findings relate to the interconnection structure of a parallel processor designed to solve this problem?

4.2 The purpose of this exercise is to show the effect of information propagation within a calculation. Use the Poisson problem of Exercise 4.1(b) and write a computer program using the Gauss-Seidel method that iterates until no interior point value changes by more than 0.1 percent. Let this be the initial state of the problem for the following exercises.

a) Increase the boundary point on the top row next to the upper left corner to a new value of 20. Perform five iterations of the Gauss-Seidel Poisson solver and observe the values obtained. Then run the algorithm until no interior point value changes by more than 0.1 percent and count the total number of iterations to reach this point.

b) Now restore the mesh to the initial state for *a*. Change the program so that, in effect, the upper left corner is rotated to the bottom right corner. To do this, scan the rows from right to left instead of left to right and scan from bottom to top instead of from top to bottom. Perform five iterations of the Poisson solver and observe the values obtained. Run the program until no interior point changes by more than 0.1 percent, and count the number of iterations to reach this point.

c) Both *a* and *b* eventually converge to the same solution because the initial data are the same and the physical process modeled is the same. However, the results obtained from *a* and *b* are different after five iterations. Explain why they are different. Which of the two problems has faster convergence? Why?

4.3 The purpose of this exercise is to examine the cyclic-reduction algorithm. Explore the solution of a one-dimensional Poisson problem by treating 15 points on a line. Let the left boundary point, point 0, have the value 10 and the right boundary point, point 16, have the value 0. Each of the 15 intermediate points has a value that is the average of its immediate neighbors.

a) Write a matrix equation of the form $\mathbf{Ax} = \mathbf{b}$ that describes this problem.

b) Simulate an iterative process that updates each interior point with the average of its neighboring points. Obtain the interior values of points for the first three iterations of the technique previously used, in which each interior point is updated by the average of its neighbors.

c) Now apply the cyclic-reduction algorithm in the text for three iterations to find one equation for the point in the middle. Solve this equation and use three iterations of back substitution to find the remainder of the points. Show your solution and the equations you obtain after each iteration. (Hint: The first iteration should produce new equations for points 2, 4, 6, 8, 10, 12, and 14. The second iteration produces new equations for 4, 8, and 12.)

d) Compare the results produced in *b* and *c* with respect to the precision obtained. Count and compare the total number of additions, multiplications, and divisions for each algorithm after three iterations.

e) Explain from an intuitive point of view why cyclic reduction yields high speed and high precision as compared to the near-neighbor iteration. What implications can you draw with regard to interconnections for processors for solving the Poisson problem?

4.4 The purpose of this exercise is to investigate how to implement conditional branches in an array computer. Program 4.1 does not show instructions that determine if convergence has been reached. The instructions should determine if every processor has obtained a satisfactory solution, and, if not, the program should branch back to the top of the loop.

a) Write the instructions that do this job, inventing the instructions as you need them. Describe the operation of each instruction that you invent.

b) Redraw the block diagram of the ILLIAC IV computer and describe the data flow on the block diagram necessary to support the test for termination.

c) Assume that the control processor of the ILLIAC IV can execute its instructions in parallel with instructions that are broadcast to the 64 numerical processors. Can any or all instructions of the termination test be overlapped with the calculation of a loop iteration? If so, describe how to implement the instructions in your program and in Program 4.1 to facilitate this overlapped execution.

4.5 The purpose of this exercise is to explore the interconnection structure of a hypercube computer such as the Cosmic Cube. Assume that you are to calculate all partial sums of *i* items up to the sum of 64 items.

a) Construct a program for a Cosmic Cube computer system that performs this operation in a time that grows as O(log N) if the number of processors is N. Assume that every node in the computer executes the same program, although the program can be slightly different from node to node since the processors in a Cosmic Cube are independent. Show explicitly the instructions that send and receive data between processors. Invent instructions as you need them and describe what the instructions do. Include some type of instruction for synchronization that forces a processor to be idle until a neighboring processor sends a message or a datum that enables computation to continue.

b) Which communication steps if any in your answer require communications with processors that are not among the six processors directly connected to a given processor? How do you propose to implement such communication in software (assuming that the hardware itself does not provide remote communication as a basic instruction)?

4.6 The purpose of this exercise is to examine the recursive-doubling solution to a linear tridiagonal system of equations. Consider the solution of the equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a tridiagonal equation.

a) Prove that the recurrence in Eq. (4.15) is a correct expression for the major diagonal of matrix $\mathbf{U}$ in an $\mathbf{LU}$ decomposition of $\mathbf{A}$.

b) Using recursive doubling, show all of the steps required to factor $\mathbf{A}$ into $\mathbf{LU}$ and to solve the equations $\mathbf{Ly} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{y}$. For each major step of the algorithm, show the basic recurrence solution. Show the mathematical formulation of your solution and indicate the basic operation in the recursive-doubling iteration.

4.7 Find a recursive-doubling technique for solving Eq. (4.13).

4.8 The purpose of this exercise is to explore some of the properties of the perfect-shuffle interconnection scheme.

a) Consider a processor that has the perfect shuffle and pair-wise exchange connections shown in Fig. 4.16. For an eight-processor system, show that the permutation that cyclically shifts the input vector by three positions is realizable by some setting of the exchange modules. Draw the network unrolled in time to show the setting that realizes this permutation.

b) Repeat *a* to show that a cyclical shift of two positions is realizable.

c) Prove that a shuffle-exchange network can realize any cyclical shift in $\log_2 N$ iterations for an N-processor system when N is a power of 2.

4.9 Find a means for evaluating a polynomial of degree $N - 1$ in the variable x in parallel on an N-processor computer that uses the shuffle-exchange interconnection pattern. Assume that N is a power of 2.

4.10 Prove that the scheme shown in Fig. 4.18 produces a sorted sequence of length N from a bitonic sequence of length N. Specifically, prove that after the comparison and exchange is performed, each sequence of length $N/2$ is bitonic and all elements of one sequence do not exceed the value of any element of the other sequence.

4.11 Consider a tridiagonal linear system such as that described in Section 4.4.4. Assume that the problem is symmetric about the major diagonal so that $a_{i,j} = a_{j,i}$. (The indices

i and j lie in the range $1 \le i, j \le N$, where N is a power of 2.) The Sturm polynomials for the matrix **A** are the polynomials $Q_i(x)$ defined by the recurrence

$$\begin{aligned} Q_i(x) &= (a_{i,i} - x)Q_{i-1}(x) - (a_{i,i-1})^2 Q_{i-2}(x) \\ Q_0(x) &= 1 \\ Q_1(x) &= a_{1,1} - x \end{aligned}$$

For a very important matrix computation it is necessary to find the number of changes of sign for a given value of x in the sequence of values $Q_0(x)$, $Q_1(x)$, . . . , $Q_{N-1}(x)$.

a) Assume that you wish to find the number of sign changes for a single value of x, and you have N processors available to do the calculation in parallel. Work out a recursive-doubling algorithm for the calculation.

b) Show a block diagram of a connection pattern suitable for this algorithm within which each processor is connected to a fixed constant number of processors regardless of the size of N, and for which at each step of the algorithm the data are accessible in a constant number of steps from neighboring processors, regardless of the size of N.

c) Now assume that you wish to find the number of sign changes for N different values of x. Compare the time taken by running your recursive-doubling algorithm N times to the time required to obtain values of the Sturm polynomials serially for each of the N values of x. Which of the two methods is preferred?

d) Now assume that you wish to compute the number of sign changes for a number of values of x much larger than the value of N. Which of the two methods is better?

4.12 Figure 4.15 shows a shuffle-exchange network with a cyclical shift interconnection pattern superimposed. Show that it is possible to compute the same set of partial sums computed in the figure without the cyclic-shift pattern, using only the perfect-shuffle and the pair-wise-exchange patterns of Fig. 4.16. Your algorithm will need to send more than one datum from one cell to a cell in the next column, but the number of different data transmitted from column to column is a constant that is independent of N.

4.13 a) Show the switch settings for a shuffle-exchange network as depicted in Fig. 4.16 that send input cell i to output cell $3i \bmod N$ for $N = 16$.

b) For each integer in the range $0 \le i \le 15$, write the value of i in binary followed by the value of $3i \bmod 16$ in binary. Start a new row for each integer and align the binary values to create a table of size 16 rows by 8 columns. Examine row i for each i. Show that the last four bits in each row are related to the switch settings from part *a*. In fact, these bits show the switch settings for input i as it passes through the network. (Hint: Use the shift-register analogy.)

4.14 a) Prove that the function that takes i into $pi \bmod N$ for $i \le N$ is a permutation when N is a power of 2 and p is odd. (Hint: The function is a permutation if you can show that when $pi = pj \bmod N$, this implies that $i = j$.)

b) Apply your reasoning from *b* of Exercise 4.13 to show that a shuffle-exchange network has switch settings that realize the permutation that takes i to $pi \bmod N$ for every odd value of p.

5

The tucked-up sempstress walks with hasty strides, While streams run down her oil'd umbrella's sides.

—Jonathan Swift, 1711

Vector Computers

The last chapter introduces the idea of building a parallel architecture matched to a specific class of problems. The discussion there mentions that there are two major models of numerical processes—a continuum model based on near-neighbor interactions and a particle model based on discrete point-to-point interactions. The major emphasis of Chapter 4 is the continuum model, together with the architectures that support processing of near-neighbor interactions for that model.

This chapter extends the discussion of numerical architectures to vector computers with the idea that these computers can be used for the majority of continuum-model problems, as well as for many particle-model problems. The vector computer has emerged as the most important high-performance architecture for numerical problems. It has the two key qualities of efficiency and wide applicability.

Most vector computers have a pipelined structure. When one pipeline is not sufficient to achieve desired performance, designers have occasionally provided multiple pipelines. Such processors not only support a streaming mode of data flow through a single pipeline, they also support fully parallel operation by allowing multiple pipelines to execute concurrently on independent streams of data.

By the mid-1980s, more than twenty manufacturers offered vector processors based on pipeline arithmetic units. They ranged from relatively inexpensive auxiliary processors attached to microcomputers to high-speed supercomputers with computation rates from 100 Mflops to rates in excess of 1000 Mflops. (One *Mflops* is 10^6 floating-point operations per second.)

The price-performance ratio of these vector processors is rather remarkable because they yield one to two orders of magnitude increased throughput for vector computations when compared to serial processors of equal cost. But this throughput increase is limited to the problems that fit the architecture—that is, to problems that can be structured as a sequence of vector operations whose characteristics make efficient use of the facilities available.

Many of the supercomputers are also high-performance serial processors for general-purpose problems, but the throughput of these supercomputers on nonvector problems is only a few times greater than the throughput of more conventional high-speed serial processors. In fact, although throughput might be high because of fast device technology, if a vector-structured supercomputer is used exclusively on nonvector problems, the computational cost may be excessive because this cost includes the cost of the vector facilities, which presumably are left idle by scalar computations.

The purpose of this chapter is to describe the general architecture of vector machines and then describe how algorithms and architecture can be matched to each other to obtain efficient processing over large classes of computations.

5.1 A Generic Vector Processor

The basic idea of a vector processor is to combine two vectors, element by element, to produce an output vector. Thus, if **A**, **B**, and **C** are vectors, each with N elements, a vector processor can perform the operation

$$\mathbf{C} := \mathbf{A} + \mathbf{B}$$

which is interpreted to mean

$$c_i := a_i + b_i,\ 0 \le i \le N - 1$$

where the vector **C** can be written in component form as $(c_0, c_1, \ldots, c_{N-1})$. The form is similar for vectors **A** and **B**.

A very simplified way to implement this operation with a pipelined arithmetic unit is shown in Fig. 5.1. The two streams of data supplied to the arithmetic unit carry the streams for **A** and **B**, respectively. The memory system supplies

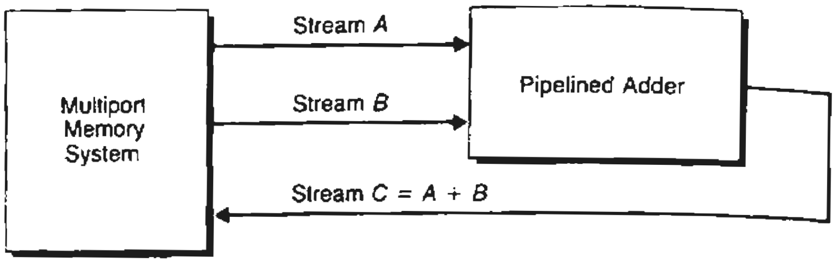

Fig. 5.1 A processor that is capable of adding two vectors by streaming the two vectors through a pipelined adder.

one element of A and B on every clock cycle, one element to each input stream. The arithmetic unit produces one output value during each clock cycle. (Actually, the input data rate need be only as fast as the output data rate. If the arithmetic unit can produce results at a rate of one output value every *d* cycles, then the input data rate need be only one input value on each stream every *d* cycles.)

برهنه، بی‌آرایش Figure 5.1 shows only the barest details of the vector processor to indicate the general flow of data through the pipelines. The pipelined arithmetic unit is discussed in Section 3.4 and that unit is the core of the architecture in Fig. 5.1.

The difficulty, however, is the design of the memory system to sustain a continuous flow of data from memory to the arithmetic unit and the return flow of results from the arithmetic unit to memory. The majority of the architectural صرف، مصروف tricks used in vector processors are devoted (اختصاص داده) to sustaining that flow of data and to scheduling sequences of operations to reduce the flow requirements.

(In this example we assume a basic one-cycle rate for the delivery of operands, production of results, and restoring of the result data into memory. This calls for a memory system that can read two operands and write one operand in a single cycle.)

قراردادی Conventional random-access memories can perform at most one READ or one WRITE per cycle, so the memory system in Fig. 5.1 has at least three times the bandwidth of a conventional memory system. Of course this ignores any additional requirement for bandwidth for input/output operations. Also, we have ignored the bandwidth for instruction fetches, but a major advantage of a vector architecture is that a single instruction fetch can initiate a very long vector operation. Consequently, the bandwidth required to fetch instructions for a vector architecture is negligible as compared to the 20 to 50 percent of the bandwidth used for instruction fetches in conventional architectures.

The major problem facing the architect is to design a memory system that
روبرو شدن

can meet the bandwidth requirements imposed by the arithmetic unit. Two major approaches have emerged in commercial vector machines.

1. Build the necessary bandwidth in main memory by using several independent memory modules to support concurrent access to independent data; or
2. Build an intermediate high-speed memory with the necessary bandwidth and provide a means for high-speed transfers between high-speed memory and main memory.

The first approach acknowledges that if one memory module can access at most one datum per access cycle, then to access N independent data in one access cycle requires N independent memory modules. The second approach produces higher bandwidth by shortening the access cycle in a small memory. But the small memory is loaded from a large memory, and the large memory can still be the ultimate bottleneck in the system in spite of the high bandwidth of the small memory.

To make best use of the small high-speed memory, we should make multiple use of operands transferred to this memory. In this way the net demand by the processor on the large memory is reduced, and bandwidth of the large memory need not be as large as the peak bandwidth required by the processor.

In the latter part of this chapter we see that another use of the high-speed memory is to provide for access patterns not available in main memory. Thus, we can move a data structure such as a matrix from main memory to intermediate memory by using the access patterns supported by main memory.

When the matrix is stored in intermediate memory, we can provide for efficient access to rows, columns, diagonals, or subarrays of the matrix, not all of which can be done efficiently when the matrix is stored in main memory. The second approach has been embellished in some cases by providing more than one level of intermediate memory, with the size, cost, and performance of each level selected to give a good cost-performance ratio of the total memory system.

5.1.1 Multiple Memory Modules

The first approach is illustrated in Fig. 5.2. In this figure main memory is composed of multiple modules. Eight modules are shown; they comprise a system with eight times the bandwidth of a single module. Each of the three data streams associated with the arithmetic pipeline has an independent path to the memory system so that each stream can be active simultaneously, provided that each individual module serves only one path at a time.

Consider how this system can be used to implement vector arithmetic. We ✓ assume that a basic memory cycle takes two processor cycles, so the bandwidth required to service the pipeline in Fig. 5.2 is at least six times the bandwidth of

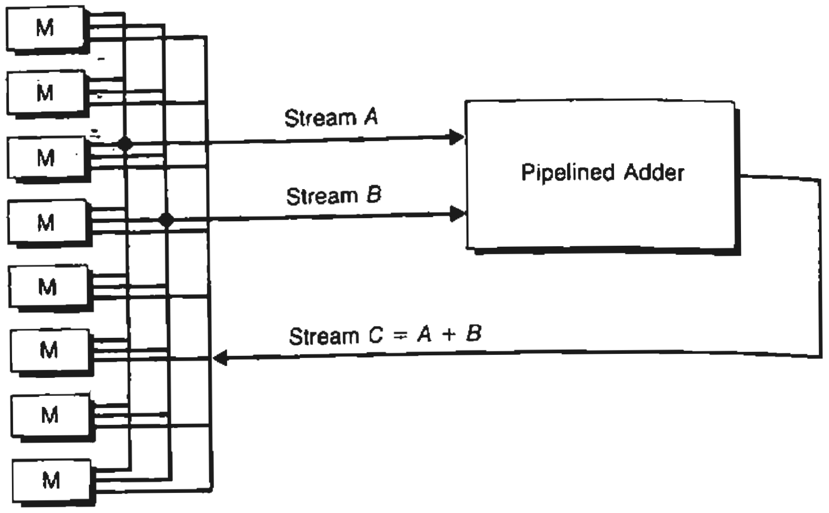

Fig. 5.2 A vector processor with a memory system composed of eight 3-port memory modules.

a single memory module) Figure 5.3 illustrates an ideal solution to our vector arithmetic example. The vectors **A**, **B**, and **C** are laid out in memory so that they start respectively in Modules 0, 2, and 4, and their successive elements lie in successive memories at addresses that are easily calculated.

The timing for the activity in this architecture is shown in Fig. 5.4. Time is shown on the horizontal axis, and the activity of the memory modules and pipeline unit is shown on the vertical axis. Note that the arithmetic pipeline has four stages, thereby producing each output value four units after the corresponding input data arrive at the pipeline. The pipeline is busy continuously after it fills with data.

A busy pipeline stage is indicated by the integer within the cell, which gives the subscript of the vector element that is being processed at the given time. A busy memory module is indicated by an *R* followed by a letter and a digit. The symbol *RA*0 indicates that the module is reading the element of vector **A** with subscript 0. The letter W indicates a WRITE operation in progress to the element of **C** whose subscript follows the W.

For this example, we have purposely allocated the vectors to modules so that no conflicts occur. To simplify this discussion we ignore the addressing of items within modules and focus only on which modules are active. At Clock 0, Modules 0 and 2 initiate READs to the first elements of vectors **A** and **B**. These elements appear at the pipeline inputs at Clock 2, and the corresponding output appears at the end of Clock 5.

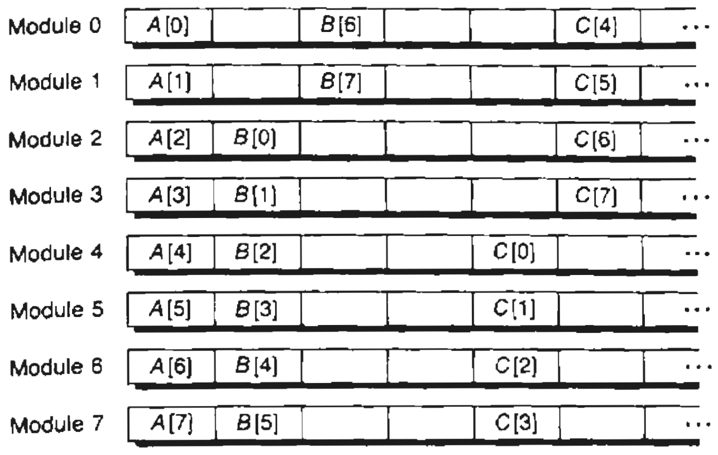

Fig. 5.3 The physical layout of three vectors in the modular memory of the pipelined vector processor of Fig. 5.2.

Meanwhile at Clock 1, Modules 1 and 3 initiate READs to the second elements of the input vectors, and at each subsequent clock cycle, successive modules initiate READs to the next elements of the input vectors. At the end of Clock 5 the first output value emerges from the arithmetic pipeline.

During the next clock period, Clock 6, Modules 5 and 6 are busy reading the next elements of the vector **A**. Module 5 delivers a_5 at the beginning of Clock 7, and Module 6 delivers a_6 at the beginning of Clock 8. Similarly, Modules 7 and 0 are busy reading b_5 and b_6, respectively, during Clock 6. Modules 1, 2, and 3 are unoccupied. Module 4 initiates a WRITE to put away c_0 during Clock 6, and during the next clock cycle, Module 5 initiates a WRITE to put away c_1.

Note how well the arithmetic and memory operations dovetail in the timing diagram in Fig. 5.4 so that all operations proceed without a collision. That is the beauty of pipelined data flow when data flows can be made collision free. But reality is never as well behaved as ideal examples are.

What happens when we cannot arrange the vectors to begin in the modules where we want them to begin? For example, the structure of the vector add prevents the vector **C** from beginning in Modules 0, 5, 6, or 7 when the input data are arranged as shown in Fig. 5.3. If **C** is computed somewhere else in the program as the sum of **D** and **E**, the vectors **D** and **E** might well be stored in memory in a way that prevents **C** from beginning in Modules 1 through 4. Hence, we might discover that **C** is too constrained and cannot be stored in any manner to support conflict-free memory operations.

Figure 5.5 shows how buffers at the input and output of the arithmetic

Pipeline Stage 4						0	1	2	3	4	5	6	7	
Pipeline Stage 3					0	1	2	3	4	5	6	7		
Pipeline Stage 2				0	1	2	3	4	5	6	7			
Pipeline Stage 1			0	1	2	3	4	5	6	7				
Memory 7						RB5	RB5	RA7	RA7	W3	W3			
Memory 6					RB4	RB4	RA6	RA6	W2	W2				
Memory 5				RB3	RB3	RA5	RA5	W1	W1					
Memory 4			RB2	RB2	RA4	RA4	W0	W0						
Memory 3		RB1	RB1	RA3	RA3									
Memory 2	RB0	RB0	RA2	RA2									W6	
Memory 1		RA1	RA1					RB7	RB7			W5	W5	
Memory 0	RA0	RA0					RB6	RB6			W4	W4		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Time (clock periods) ⟶

Fig. 5.4 A timing diagram for the addition of two vectors, component by component, in pipeline mode.

حذف، رفع مبارزه

pipeline can eliminate contention at the memory. Suppose, for example that all vectors start in Memory 0. The timing diagram in Fig. 5.6 shows how the vector operation proceeds without conflict. The input buffer on the **A** input is set to a delay of two clocks, and the output buffer is set to a delay of four clocks.

In Fig. 5.6 note that **A** is read before **B**, so that each element of **B** reaches the pipeline exactly two clocks after the corresponding element of **A** emerges from the memory. By buffering **A** for two clock cycles, we provide for corresponding elements of **A** and **B** to reach the arithmetic pipeline concurrently.

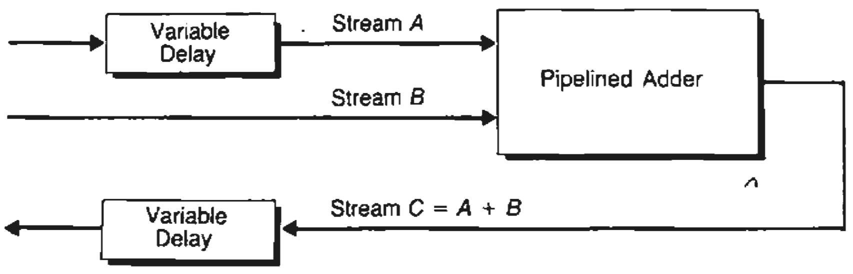

Fig. 5.5 Variable delays in the input and output streams of a pipelined arithmetic unit.

	0	1	2	3	4	5	6	7	8	9	10	11	12
Pipeline Stage 4								0	1	2	3	4	5
Pipeline Stage 3							0	1	2	3	4	5	6
Pipeline Stage 2						0	1	2	3	4	5	6	7
Pipeline Stage 1					0	1	2	3	4	5	6	7	
Memory 7								RA7	RA7	RB7	RB7		
Memory 6							RA6	RA6	RB6	RB6			
Memory 5						RA5	RA5	RB5	RB5				
Memory 4					RA4	RA4	RB4	RB4					
Memory 3				RA3	RA3	RB3	RB3						
Memory 2			RA2	RA2	RB2	RB2							
Memory 1		RA1	RA1	RB1	RB1					RA9	RA9	RB9	RB9
Memory 0	RA0	RA0	RB0	RB0					RA8	RA8	RB8	RB8	W0

Time (clock periods) ⟶

Fig. 5.6 A timing diagram for the addition of two vectors when storage conflicts arise. After reading, Vector **Fig. A** is delayed by two clocks, and, before writing, Vector **Fig. C** is delayed by four clocks. The first WRITE takes place at Clock 12.

When the first result appears at the output of the pipeline at the end of Clock 7, it arrives just when Module 0 is busy for four clock cycles fetching a_8 and b_8.

Hence, the output buffer holds each output for four clock cycles and then passes the output to the memory system. Thus the first result is stored during Clock 12, and the total duration of the vector operation is lengthened by six clock cycles over the timing shown in Fig. 5.4. After the initial delay, however, results are produced and stored at the rate one result per clock cycle, which is the same rate as in Fig. 5.4. The technique of adding buffers to the inputs and outputs of an arithmetic unit to eliminate memory conflicts is similar in spirit to the idea of adding buffering in the interior of a pipeline to eliminate internal conflicts, which has been explored earlier in Section 3.4.4.

One implementation of this idea is shown in block diagram form in Fig. 5.7, which is intended to represent the structure of the CDC STAR Computer, a supercomputer produced in the mid-1970s. This diagram shows a variable delay inserted into one of the operand streams and the result stream. The delays are set to specific values depending on the location of the first elements of each of the operands and the result vector. This ensures that the pipeline can run at full speed after an initialization period during which the operand and result streams fill their respective buffers. Unfortunately, if vectors are short, a relatively long buffering delay can have a strongly negative influence on performance.

Figure 5.7 shows that several functions can be selected within the arithmetic

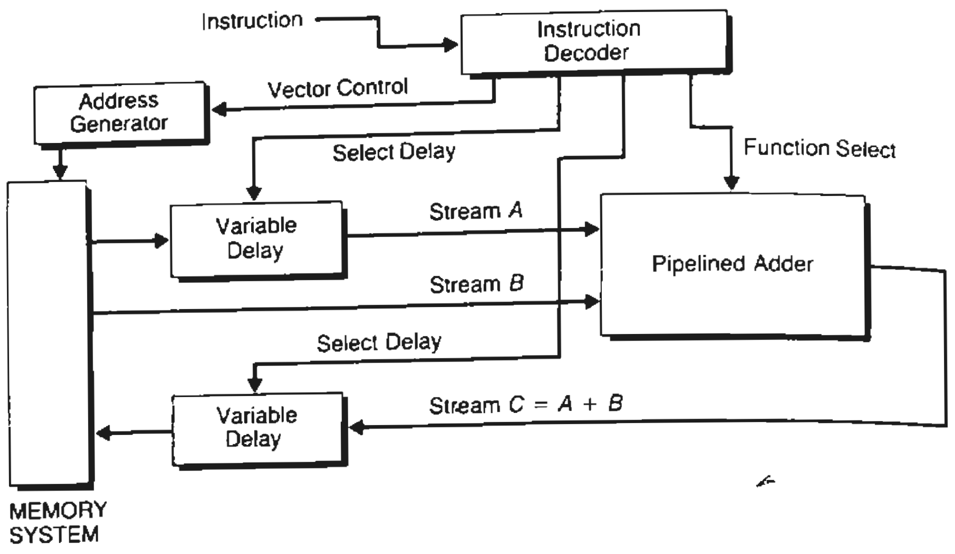

Fig. 5.7 An architecture similar to the CDC STAR. The instruction decoder sets the variable delays as a function of the starting addresses of the vectors and the throughput rate of the arithmetic pipeline for the specified operation. The address generator produces the load and store addresses during the execution of the instruction.

subsystem. The CDC STAR has no capability to overlap two or more vector operations with each other, so it is reasonable in this architecture to share common arithmetic functions among different vector operations. Thus the floating-point addition and multiplication operations use the same hardware for exponent add, shift, and mantissa add, which are common to the two functions. The CDC STAR actually provides for two single-precision operations or one double-precision operation within one pipeline, where the flexibility is obtained by special logic inserted in the arithmetic stages that lie in the boundary region between the two single-precision halves of a double-precision operand. This logic disables the carries between halves in 32-bit mode and enables the carries between halves in 64-bit mode. This permits the result rate for single precision to be double the result rate for double precision, when you measure the result rate in terms of result operands produced per unit time. However, the number of physical bits produced per unit time is the same for single and double precision.

The variable delays in Fig. 5.7 are rather interesting entities in themselves because they can be costly both in dollars and setup time. Even if the dollars are unimportant, setup time is very important, and we require the delay to be set quickly to a particular value.

One possibility is to use a tapped delay line wherein the data stream enters

[illegible]

a series of delay stages at a specific input, but a tap control selects a specific output to serve as the output of the delay line. This is shown in Fig. 5.8. Each of the N stages in this delay line is a potential network output, but the actual network output is determined by the output control.

This line can yield any delay from 0 to $N - 1$, provided that data can be clocked in and out of the delay line within a single clock cycle. In some technologies, the logic required to implement the variable delay results in relatively long access paths that may be too long for the clock cycle of the full system. This is technology dependent, however, it must be considered by the architect.

An alternative way to achieve the variable delay is shown in Fig. 5.9. This requires N cells of a special memory. This particular memory can simultaneously read any cell in the system and write any other. There are two address registers, one for READ and one for WRITE. The initial value of the WRITE register is 0, and as each datum arrives at the memory and is written, the WRITE address increments by 1.

To achieve a delay of an arbitrary amount up to N, the initial address of the READ register is $-d$, the selected delay. This register is incremented at the rate of operand arrivals, but no data are read until the READ address is 0. At this point the READs occur at the same rate as the WRITEs, and thus the output stream is the same as the input stream shifted d units in time.

The memory in Fig. 5.9 has exactly N locations, numbered 0 to $N - 1$. As READ and WRITE addresses to memory increment beyond $N - 1$, they reset to 0 and continue incrementing, so the memory operates as a circular queue. The value of N need only be large enough to provide for the longest delay required for synchronization. Vector operands can be much longer than N be-

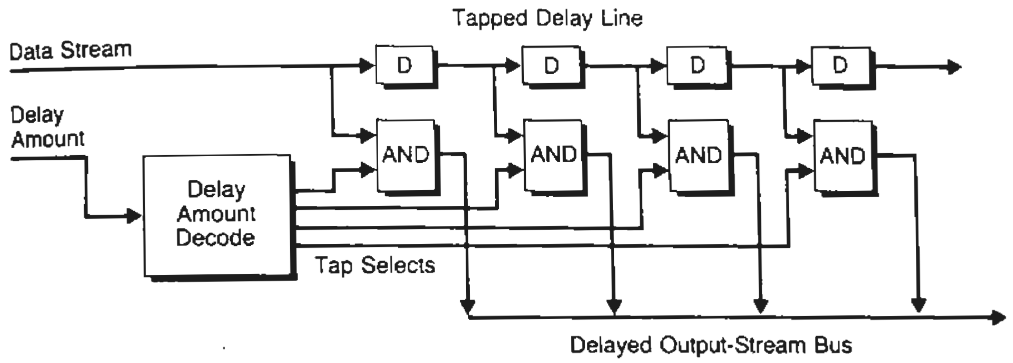

Fig. 5.8 A variable delay built from a tapped delay line. The D modules are unit delays. One tap is gated to the output bus by a tap-select control line produced by decoding the delay amount.

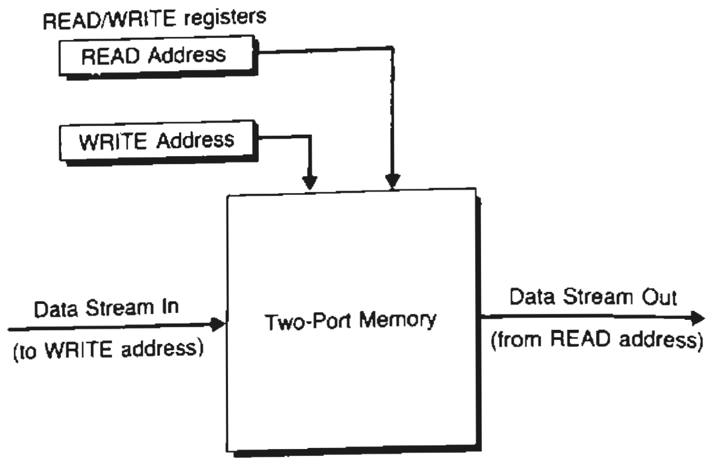

Fig. 5.9 A variable delay implemented with a two-port memory. The delay is the difference between the READ and WRITE addresses. For 0 delay, the input stream is shunted directly to the output by means of bypass logic not shown in the figure. [illegible]

cause the delay memory does not have to store an entire vector at any given instant of time.

The delay 0 case is a special situation that can easily be detected because the READ and WRITE addresses are identical in this case. In this situation the input data stream must be shunted directly to the output without being stored in the buffer. Interested readers will find more discussion on variable delays in Kogge [1981].

The variable delay memory in Fig. 5.9 is capable of delaying a stream any amount from 0 to N clock cycles. It has several advantages over the tapped delay-line because no more than two addresses in Fig. 5.9 change state each cycle, as compared to changes in potentially all stages of a tapped delay-line. Each time a cell changes state, there is a change in a physical parameter such as voltage or current. Each such change usually requires power, and with power is produced heat and electrical noise. The fewer changes in the memory system of Fig. 5.9, as compared to the delay memory of Fig. 5.8 in which many cells change on each clock cycle, lead to potentially fewer transient effects and noise problems.

5.1.2 Intermediate Memories

We indicate earlier that an alternative to providing high bandwidth in main memory is to provide one or more intermediate levels of memory to form a hierarchy of memories, with the highest bandwidth memory placed closest to the processor. In this architecture, vectors migrate from main memory to the fastest memory in the hierarchy as they are needed by the processor. Other memory levels, if they exist, provide intermediate storage points to hold vectors in transit just before or just after their use in the fastest portion of the hierarchy.

The Cray I, a landmark high-speed architecture, bases its high-speed operations on a hierarchical memory structure. A simplified diagram of the Cray I appears in Fig. 5.10. Its main memory (8 M-bytes) is separated from the processing units by one or two levels of intermediate memories. For vector operations, the intermediate memory is a set of eight vector registers (the *V* registers), each capable of holding a 64-element vector of double-precision numbers. The vector pipelines obtain data from the vector registers, not from main memory. Similarly, the result vectors from the pipelines are returned to the vector registers.

Scalar operands have two levels of intermediate memory, much like conventional cache-based high-performance systems. The fastest level contains eight 64-bit scalar registers (the *S* registers), which communicate directly with the pipeline units for scalar arithmetic.

A slower, but still very high speed, level of intermediate memory is composed of 64 scalar registers (the *T* Registers), each 64 bits in length. The *T*-register scalar memory has the same purpose as a cache memory in that it is intended to hold those data that overflow from the high-speed scalar registers. Such data may become idle temporarily, but should be held close to the processor

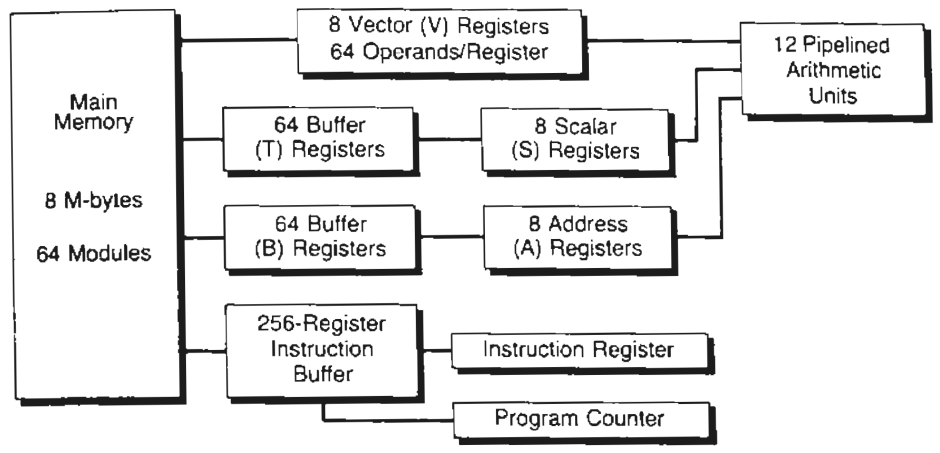

Fig. 5.10 The Cray I—an architecture based on hierarchical memories. One to two levels of high-speed intermediate memories isolate the arithmetic and instruction logic from main memory.

in anticipation of future need rather than moved to the more remote main memory between periods of use. Also, new data can be prefetched to the intermediate scalar memory from main memory just prior to use in the arithmetic unit.

Unlike a cache memory, this intermediate memory is not managed automatically. Data must be transferred explicitly to and from the intermediate memory by means of ordinary program instructions. The disadvantage of this scheme over cache memory is that the Cray I intermediate memory has to be managed by the programmer or the compiler. The big advantage of this type of memory over cache memory is speed—intermediate memory is accessed by means of physical register addresses, not by a cache lookup. The cache lookup tends to take longer because a cycle must be long enough to support both the normal read operation plus an address comparison, whereas the Cray I intermediate memory does not require the time to compare address tags in a cache.

Cray designs usually provide for short high-speed registers to hold addresses, and the Cray I follows this general philosophy. It has eight address registers (the *A* registers), each 24 bits in length. These are backed up by an intermediate level of memory in the form of 64 registers (the *B* registers), each 24 bits in length. Thus the *B* registers function as a cache for the *A* registers, except that all operations on the *B* registers are explicitly controlled by program instructions rather than automatically controlled, as are the registers of a cache memory.

One more intermediate-level memory appears in the diagram. This is an instruction buffer that holds portions of the instruction stream that are fetched just prior to the execution of those instructions. Tight inner loops tend to lie completely within the instruction buffer and can execute repeatedly without requiring fetches to main memory. Because many applications written for the Cray tend to spend the great majority of time in tight loops, instruction fetches tend to be rather rare events.

Note in Figure 5.10 that every functional portion of the processor has a high-speed memory attached to it. No function is directly attached to main memory, as is the case for the processor structure shown in Fig. 5.7. Moreover, some of the high-speed memories are backed up by memory buffers that lie between main memory and high-speed memory.

The structure of the design clearly shows the major idea of the architecture—keep the processing units busy by keeping their operands close at hand. The intermediate memories represent a compromise in the sense that they provide a pool of data readily accessible to the processing units at lower cost than the cost of storage in the fastest levels of the memory hierarchy.

The performance of the intermediate memories is, however, below the performance of the highest-speed memories. To design such a hierarchy involves comparing the performance trade-offs, with and without intermediate memory, and the savings attributed to using intermediate memory in place of high-speed registers. Note that the savings is partly due to cost and partly due to decreased volume and power consumption, which may be the deciding factors in supercomputer design.

An intermediate memory can also provide a buffer for reformatting data structures for efficient processing. The idea is that the pipeline is optimized for access to successive elements from a vector register, but the items to be processed need not lie in consecutive cells of memory. The operands can be fetched into an intermediate memory and from there sent to the vector registers. In so doing, the operands can be reorganized so that the items to be processed next are moved to contiguous cells of a vector register. Methods for making this transformation are covered in more detail in the next section.

The most distinguishing feature of the two architectures described in this section is in regard to coupling operand memory to the pipeline. The first architecture relies on main memory to hold pipeline operands, so main memory must have a bandwidth at least as large as is required by the arithmetic unit. This forces all of main memory to either be fast or partitioned into many independent memory modules, or both, because the peak bandwidth requirement of the arithmetic unit is very high.

The second design provides for the very high bandwidth to be supplied by a register memory much smaller than main memory, and thus, the slower speed of main memory need not handicap the arithmetic pipeline. Another facet of the second design is that it provides for the possibility of overlapping pipeline operations because the gross bandwidth of the high-speed registers can be made high enough to meet peak processing requirements of several pipelined arithmetic units combined.

The cost of providing extra bandwidth for the registers is the cost of providing extra ports for reading and writing the registers. While this cost can be relatively high per bit of storage, the high-speed registers have only 10^4 to 10^5 bits, as compared to the 10^8 to 10^{10} bits of main memory. Thus, it is feasible to supply extra ports to the registers but impractical to do so for main memory.

The Cray I does provide for overlapping pipelined arithmetic operations so that as many as three independent vector operations can be done concurrently. A vector operation produced on one output stream can be routed directly to the input of the next operation. The first architecture has no provision for additional data streams, so the result stream has to be stored in memory before it can be rerouted to an arithmetic pipeline for additional processing.

Because the variable delay is shared by all vector operations, the buffer in the variable delay has to empty before the delay can be reset for the next pipelined operation. Hence the pipeline must drain between operations, and no overlap is possible. The Cray I's ability to overlap pipelined operations is strictly due to its intermediate buffers and high-speed registers.

In our discussion of cache memory, our assumption is that cache memory is an extremely important architectural feature of high-speed computers. Yet the

Cray I has no cache-organized memory, although it does have several memories that occupy a place in a memory hierarchy similar to the place of cache memory. The absence of cache is due partially to design decisions and partially to characteristics of vector programs that may differ from the characteristics of scalar programs.

The design decision for this class of machine has to weigh the cost and difficulty of programming an intermediate memory that is not cache organized against the performance penalty for a cache access as compared to a register access. The Cray I is built for performance. Its users are rather sophisticated and are willing to expend extra effort in software to obtain a performance boost. This biases design decisions against the use of cache and toward the use of programmable registers.

Moreover, a cache may not work as well for vector operations as it does for scalar operations, although currently there is very little experience on which to make a judgment. The designer has to consider these questions:

- How large should a cache be on a vector machine?
- Should it be large enough to hold a few full-length vectors?
- Or should it be smaller and instead hold fragments of many different vectors?

These questions are largely unanswered today, but we can expect them to be explored in the next few years as vector technology becomes more mature and implementers seek methods for boosting performance of machines built today without caches.

Serial access to vectors dictates against a cache that uses LRU replacement because one vector load may flush an entire cache and leave only dead data in the cache. Perhaps a cache organized to manage vectors may be useful, but this is still a matter of conjecture and needs further study. Therefore, vector registers should be organized as program-accessible registers rather than as a cache until performance studies show how to improve throughput with a vector-organized cache.

The various intermediate registers, including the *T* (scalar) registers, the *B* (address) registers, and the instruction buffer, are the most obvious candidates for cache organization. The hit ratio should be comparable to the hit ratio for conventional serial machines if these registers were cache organized, but interlocks across the caches to maintain consistent data would be a serious problem.

Several units in the Cray I can modify data. Any such modification has to be reflected in a cache that holds copies of such data. Cache consistency requires that each time a new item is modified in cache, a cross check is made at all other caches to see if the same item is contained there. This could hurt performance by causing conflicts for cache access.

Although this implementation is not the only way to interlock cache access, interlocking is almost always accompanied by a reduction in performance and possibly by a modest increase in cost. So cache may well be unattractive for a Cray-type environment.

Future designs, however, need not follow the directions of the Cray I. Device technology can change dramatically, resulting in different available densities, speeds, and costs of memory. Major changes in any or all of these factors could produce vastly different architectures. As memory becomes smaller, faster, and less expensive, there is a potential for intermediate memories of much greater capacity. Higher power densities, however, may require that volumes be held small to enable the computer systems to be cooled and may force the designer to resort to small intermediate memories or elect not to use them in some areas of the design. A reasonable rule of thumb in the supercomputer area is to build as much capacity and performance capability as possible, and then look for ways to reduce volume, power consumption, and total cost without drastically hurting performance.

5.2 Access Patterns for Numerical Algorithms

High performance requires that the architecture fit the workload. A high-speed machine must do the job for which it is intended. Although the discussion in the previous chapter cautioned against structures that are too special purpose, we must at least understand the requirements for a large class of problems to make sure that we can solve those problems effectively.

If design compromises are necessary, then we should understand a pure design with no compromises and then evaluate the compromises separately. In this section we examine some numerical problems and learn that access patterns play a critical role in determining the execution speed of the algorithms. We show how to build machines that support the special access patterns frequently encountered in large numerical calculations.

Heller's excellent review of parallel algorithms for numerical methods [1978] focuses on linear algebra because most large-scale practical applications of numerical methods are expressed in terms of matrices and vectors. This is not surprising; matrix notation gives a compact way to express enormous amounts of computation.

Consider two extremes for writing a program that performs 10^{10} multiplies. At one extreme, the programmer writes a few hundred or thousand lines of program statements, many of which are just calls on a library of matrix and vector functions. At the other extreme, the programmer is faced with solving an unstructured problem and has to specify each of the 10^{10} lines individually.

It is quite clear that no one will write the latter code—it takes an extraordinary amount of time to write. At the rate of one arithmetic operation per second, a person working full time would need 30 years to write down all of the arithmetic expressions that describe the workings of the program. A computer that executes at 100 Mflops takes only 100 seconds to execute that program.

Obviously, vector and matrix operations are very important for a high-speed architecture because many very large algorithms are expressed succinctly by such operations. The demonstrated importance of numerical applications for large-scale computations leads us to treat the world of vector and matrix computations in this chapter.

Other notational systems may also be useful. For example, recursively defined functions are succinct descriptions of potentially massive computations. In any case, we are unlikely to generate unstructured large-scale computations simply because the programming effort to write such applications is unreasonable.

5.2.1 Gaussian Elimination

Heller [1978] covers a number of algorithms for solving linear systems of the type

$$\mathbf{Ax} = \mathbf{b}$$

where **A** is an $N \times N$ matrix, and **x** and **b** are $N \times 1$ column vectors. The objective is to find **x**, given **A** and **b**. The techniques available depend on the specific characteristics of the matrix **A**.

When **A** is dense, that is, when all or nearly all of the components of **A** are nonzero, the solution of the linear system of equations can be found by carrying out a succession of row and column operations on **A**, with corresponding changes made to **b** during the course of the computation.

One efficient and effective method of solution, Gaussian elimination, factors **A** into the product of two triangular matrices, **L** and **U** where **L** is lower triangular and **U** is upper triangular. We see this in the previous chapter for the special case in which **A** is tridiagonal, and **L** and **U** are bidiagonal. In both the general and the special case, the factorization must compute the elements of **L** and **U**, and this is possible to do by means of operations on row and column vectors.

Once the factorization produces **L** and **U**, the next steps solve the triangular systems

$$\mathbf{Ly} = \mathbf{b}$$

and

$$\mathbf{Ux} = \mathbf{y}$$

to obtain a value of **x** that satisfies the original equation since

$$\mathbf{Ax} = \mathbf{LUx} = \mathbf{Ly} = \mathbf{b}$$

The solutions to the triangular systems are particularly easy to obtain by means of vector operations on rows of the **L** and **U** matrices.

When **A** is developed from partial differential equations that describe a problem in the continuum, **A** is a sparse, highly regular matrix whose solution can be determined quite efficiently using techniques such as cyclic reduction, which is described in the previous chapter. Although we may view such a matrix **A** as being composed of a collection of row or column vectors, the nonzero components of **A** in problems that arise from continuum formulations tend to lie only along a few diagonals. Many algorithms approach the solutions of this type of sparse-matrix problem by treating the matrix as composed of diagonal vectors, so that vector operations manipulate streams of data fetched from various diagonals of the **A** matrix.

It is worthwhile to examine in detail one example of a parallel algorithm for computing the solution to a linear equation. In this case, we look at classical Gaussian elimination and assume that the basic parallel operation can manipulate a row or column of **A** in equal time. This assumption is not true for all architectures, and its correctness requires some resourcefulness from both the computer architect and the numerical programmer. Nevertheless, let us assume that rows and columns are equally accessible and explore how to create an algorithm from row and column operations.

The core of the algorithm produces a new column of **L** and row of **U** at each of N iterations. The new data for **L** and **U** overwrite corresponding locations of **A** and are unchanged for the remainder of the computation. Before producing the next elements of **L** and **U**, the algorithm updates the entire portion of the **A** matrix that has not yet been overwritten. The diagonal of **L**, which is forced by this algorithm to be all 1s, is not stored explicitly. The diagonal of **A** is eventually overwritten by the diagonal of **U**.

At each iteration, one diagonal element of **A** is overwritten. We call this element the *pivot* for that iteration. In the matrix below the pivot is stored the new column of **L**, and to the right of the pivot is stored the new row of **U**. Figure 5.11 shows the various portions of the data at the start of an iteration. The L and U denote the columns of **L** and rows of **U** that have been computed up to this point. The P designates the pivot. The L' and U' denote the new data to be computed during this iteration, and the A denotes the elements of **A** that will be transformed during this iteration.

For numerical stability, we should choose as the pivot the element with the greatest magnitude in the region that includes P, L', U', and A. If this element is not P, then that element can be brought to position P by a swap of rows and columns. Most algorithms, however, do not search such a large area for the new pivot.

The algorithm remains stable, although it has a larger error bound, if the pivot element is the largest element in the area that includes P and L'. If the largest element is not in position P, then by exchanging the row containing the element and the pivot row, we can move the large element to position P. Row and column exchanges are permitted because they do not change the

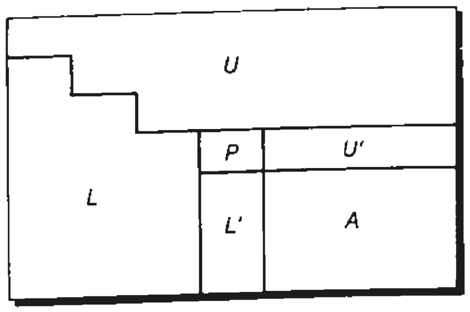

Fig. 5.11 The regions of a matrix revealed during a single cycle of an LU-decomposition algorithm for performing Gaussian elimination.

solution to the original system of equations, although the elements in the solution vector in general will have to be permuted to produce a solution vector whose elements are ordered correctly in regard to the original problem.

Program 5.1 is a simplified version of an algorithm for Gaussian elimination that appears in Forsythe and Moler [1967]. This algorithm is expressed in vector notation, where the notation $A[i,j]$ designates a single element $a_{i,j}$, of **A**, and $A[1 \ldots j-1,j]$ designates a column vector of **A**. In this case, the subscript range $1 \ldots j-1$ designates all subscript values lying between 1 and $j-1$. The single subscript in the second component designates the jth column. Hence, $A[1 \ldots j-1,j]$ is the vector that consists of the first $j-1$ elements of the jth column of **A**. The same notation holds for rows, except that the subscript range is placed in the second subscript position.

The important aspects of this example are that:

1. The algorithm as expressed accesses both rows and columns.
2. The majority of the vector operations have either two vector operands or a scalar and vector operand, and they produce a vector result.
3. The MAX operation on a vector returns the index of the maximum element, not the value of the maximum element.
4. The length of the vector of items accessed decreases by 1 for each successive iteration.

The first point is consistent with our assumption that we need to access both rows and columns in some algorithms. It turns out in this problem that the inner loop can be done either by rows or by columns; the choice is up to the programmer. But the algorithm does require both a column and a row operation elsewhere, so a vector computer should provide easy access to both rows and columns, at least, and possibly other interesting forms of access.

Program 5.1 Gaussian elimination.

FACTOR is a vector algorithm for factoring matrix **A** into **L** and **U**, where **A** = **LU**, **L** is lower triangular, and **U** is upper triangular. The diagonal elements of all matrices are equal to 1; they are not stored explicitly. **L** overwrites the lower triangular portion of **A**, and **U** overwrites the diagonal and upper triangular portion of **A**.

```
for i := 1 to N do
  begin {* Search Column for a pivot element. *}
        {* Find the index of the element with the largest absolute value in the pivot
           row. *}
        imax := index_of_Max(abs(A[i ... N,i]));
        {* Swap Row imax with Row i. This produces a new row of U. *}
        Swap(A[i,i ... N],A[imax,i ... N]);
        {* Check for singularity, and terminate if so. *}
        if A[i,i] = 0 then singular matrix;
        {* Find the new column of L, and store it in A. *}
        A[i+1 ... N,i] := A[i+1 ... N,i]/A[i,i];
        {* Update the remaining part of the A matrix. *}
        for k := i+1 to N do
          A[k,i+1 ... N] := A[k,i+1 ... N] − A[k,i] × A[i,i+1 ... N];
  end; {* Outer loop *}
```

The next point indicates that a vector pipeline should provide a mechanism to have a scalar serve as one of the operands, and in so doing it should produce an answer faster or more efficiently than a similar operation that has both operands as vectors.

The third point suggests that the pipelined arithmetic unit should provide some mechanism for producing results that are scalar, such as results produced by the functions MAX, MIN, and SUM. Note as well that the scalar result might be an index of an important element in the vector and not necessarily the value of a vector element or of a combination of vector elements. In our example, the information required by the algorithm is the index, not the matrix element.

The last point is the most perplexing. The vectors used by this algorithm shrink with each step, and thus the last step uses vectors of length 1. Pipelined arithmetic and vector operations have a certain overhead, and we should attempt to amortize that overhead over many operands by treating long vectors as much as possible. We have an efficient machine if the overhead for starting a vector computation is small compared to the amount of useful work it produces. However, if the useful work produced by a vector operation is very small, the overhead may be painfully expensive and drastically reduce the efficiency of the system. The last point forces us to keep vector overhead as small as possible because we inherently must deal with short vectors for some portions of important computations.

The next section illustrates some techniques for solving the access problem and gives insight into the structure of efficient vector processors.

5.3 Data-Structuring Techniques for Vector Machines

In this section we explore the problem of accessing data in ways that are constrained by an algorithm. If a data structure such as a matrix is to be accessed only by rows, we can store rows so that consecutive elements lie at successive addresses. If only columns of a matrix are required, we could store the matrix in a column-oriented fashion, by putting consecutive elements of each column at consecutive memory addresses. But if both row access and column access were required, there is no obvious way to meet both constraints efficiently.

The problem is illustrated in Fig. 5.12, in which a matrix is stored in a main memory composed of eight independent memory modules. The modules are represented as columns. In Fig. 5.12(a), an 8 × 8 matrix is stored so that its row elements can be accessed in a pipeline fashion. Each successive row element is stored in the next memory module.

✓ (If a memory access takes several clock cycles, this memory can still deliver one row element per clock cycle after an initial delay) To fetch the row vector for Row 0, for example, initiate a fetch to the (0,0) element, and before this element is delivered to the memory bus, initiate a fetch to the (0,1) element on the next clock cycle. On Clock i, initiate a fetch to element $(0,i)$.

If the memory access time produces a delay d between the initial access to an item and the time at which it appears at the memory output port, then in our example the element $(0,i)$ can be placed on the memory bus at the end of Clock $i + d$. This is the method of overlapped access described at the beginning of this chapter. If d does not exceed 8, the number of distinct memory modules in the example, the vector can be arbitrarily long. If d is greater than 8, however, attempts to access vectors longer than eight result in collisions at some memory module because the module is asked to initiate a fetch for a new element before its access to an old element has been completed.

Another way to describe the situation is that the memory bandwidth must be great enough to support the memory demand. If the delay d is greater than 8, then the aggregate bandwidth of the eight memories is less than one item per clock period, yet the pipeline demand is for one item per clock period. With delay d, the aggregate bandwidth is one item per clock period only if there are at least d independent memory modules, each capable of accessing one item per d clock periods. If an instruction requires three streams, two for input operands and one for results, then the aggregate bandwidth of memory must be at least

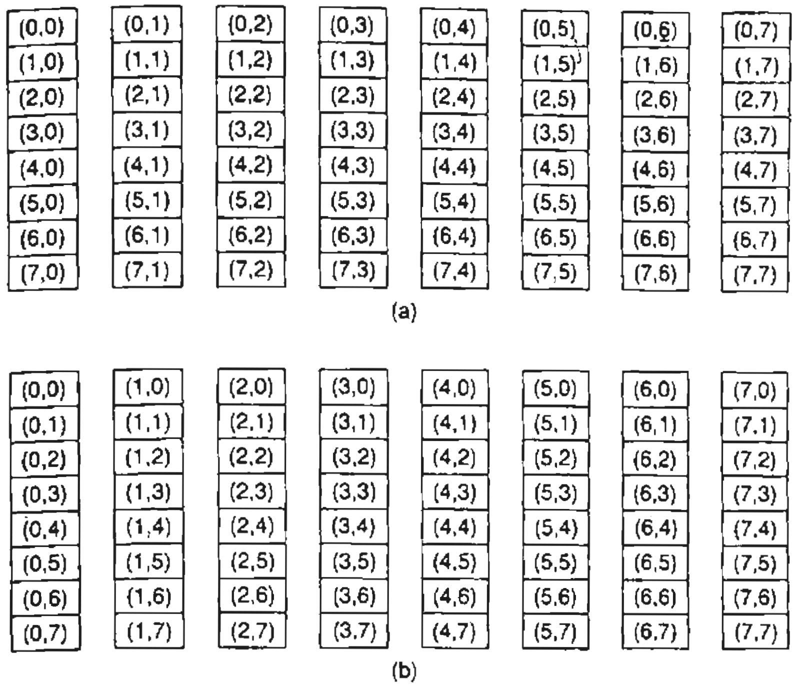

Fig. 5.12 Two of several possible storage formats for an 8 × 8 matrix: (a) Suitable for access to row vectors, but bad for column vectors; and (b) Suitable for access to column vectors, but bad for row vectors.

three items per clock period, so the number of memory modules must be at least 3*d* to support a pipeline rate of one result per clock time.

Figure 5.12(a) shows that the memory bandwidth available is not the whole story. Consider what happens if you need to access columns of the matrix, for example Column 0. In this figure, Column 0 lies wholly in one memory module. No matter how many other modules are in the system, access to the elements of Column 0 is limited by the maximum bandwidth of the single module. In this case, at most one item can be delivered every *d* units of time, and it is impossible to support a rate of one column element accessed per clock period unless one module by itself can produce data at this rate—that is, unless *d* is unity.

In Fig. 5.12(b) we transpose the matrix to give fast access to columns, which are now stored across the memories, but we give up fast access to rows. The Gaussian elimination algorithm, as reproduced in the previous section, requires both row and column access, so neither the storage pattern of Fig. 5.12(a) nor Fig. 5.12(b) is acceptable. One way to circumvent the problem is to rewrite the

[illegible]

algorithm to use column or row access exclusively. This happens to be possible for Gaussian elimination, but it is not always possible to revise an algorithm to live within the access constraints of memory.

Another approach is to alter the structure of data in memory. Figure 5.13 shows the same matrix stored so that successive rows are skewed with respect to the previous row. In this case Row 0 starts in Module 0, Row 1 starts in Module 1, ... , with each row shifted to the right by one column with respect to the immediately preceding row.

In this storage scheme the address of an item in a system address space is 8 × (local address) + module number, where each individual memory has a local address-space, and the module numbers range from 0 to 7. Row elements lie at successive addresses in the system address-space. Successive column elements lie at addresses that differ by nine in system address space. Note that successive column elements lie in different memories in this system, and that they can be accessed in pipeline fashion as efficiently as successive row elements.

Even though the matrix is 8 × 8, we store the matrix as if it were 8 × 9 (8 rows by 9 columns), wasting the memory allocated to the ninth column. The extra column provides the cyclical offset of successive rows, so column elements are spread across all memories just as row elements are.

To use this storage structure in a vector processor similar to those shown in Figs. 5.7 and 5.10, the vector operand must be specified by four quantities:

1. Starting address;
2. Number of elements;
3. Precision (number of bits per element); and
4. Stride (offset between successive elements).

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(1,7)		(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(2,6)	(2,7)		(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(3,5)	(3,6)	(3,7)		(4,0)	(4,1)	(4,2)	(4,3)
(4,4)	(4,5)	(4,6)	(4,7)		(5,0)	(5,1)	(5,2)
(5,3)	(5,4)	(5,5)	(5,6)	(5,7)		(6,0)	(6,1)
(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)		(7,0)
(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	

Fig. 5.13 A data structure that permits access to both rows and columns. Row access has stride 1. Column access has stride 9. The blank entries in the matrix form a dummy ninth column of the 8 × 8 matrix.

✓ The *stride* for a vector expresses the address increment used to move from one element to the next in a vector access. The stride for row access in Fig. 5.13 is 1, and the stride for column access is 9. (In general, if the stride is relatively prime to *M*, the number of memories, then *M* successive accesses for that stride are directed to *M* distinct memories. More generally, for any *M* and stride *s*, *M* successive accesses of stride *s* are directed to *M*/GCD(*s*, *M*) different memories, where GCD is the greatest common divisor function. GCD is equal to unity, by definition, when its arguments are relatively prime.)

(Since *M* is usually a power of 2, this is equivalent to saying that any vector access with an odd stride produces *M* consecutive accesses to *M* distinct memories. In Fig. 5.13, one can easily verify that 11 × 11 and 13 × 13 matrices support row and column access as readily as the 9 × 9 matrix. For column accesses, address conflicts arise when a matrix has an even number of columns because even numbers are not relatively prime to *M*. For example, a 12 × 12 matrix causes problems when *d* exceeds 2 because column elements 1, 3, 5, ... , all lie in the same memory module. For a similar reason, 8 × 8 and 24 × 24 matrices lead to the same inefficient access to columns.)

✓ (Fortunately, for every even number the next number is odd, so for every bad value for a number of columns, the next larger number is good. Hence, we can always add a wasted column to a data structure and provide a storage structure that is ideally suited to pipelined row and column access.)

If row and column access were the only requirements, our discussion would end here. But the designer should not limit a design to a small class of problems. If a few changes can greatly increase the number of problems that can run efficiently, we must explore those changes and the consequences of making them.

Kuck's study of parallelism [1976] (*see also* Budnik and Kuck [1971]) suggests that typical access patterns to matrices include access to:

- Matrix diagonals in the major and minor directions;
- Square subarrays; and
- Rows and columns.

Note that the stride required to access the major diagonal of a matrix is one greater than the stride required to access a column of a matrix. If *M*, the number of memory modules, is a power of 2, then column access and major diagonal access cannot both be efficient since one stride or the other is not relatively prime to *M*.

Budnik and Kuck [1971] make a startling suggestion—use a number of memories that is not a power of 2. For example, if the number of memories is a prime *p*, then all strides less than *p* are relatively prime to *p*. Therefore, we can store arrays in a structure that yields equally efficient access to rows, columns, and diagonals. Budnik and Kuck explore this notion in the context of a parallel computer that is fully parallel in access, rather than pipelined. This notion was developed further by Burroughs in the design of an unusual supercomputer called the BSP (Burroughs' Scientific Processor), whose structure is shown in Fig. 5.14.

The BSP design provided for 17 memories, rather than 16, to solve the problem of supporting all interesting ways to access a matrix. Memory is not pipelined in this architecture. Rather, in one memory cycle the memory system delivers one block of 17 memory lines, each line from a distinct memory. Two networks separate the 17 memories from 16 processors. The input alignment network shrinks a 17-way access to 16 operands by deleting some operand and compressing the remaining 16 operands into a contiguous vector.

This process is shown in Fig. 5.15 in simplified form for compressing a five-way vector read to deliver data to four processors. Figure 5.15(a) shows access to a column of a 4 × 4 matrix, and Fig. 5.15(b) shows access to a diagonal of the same matrix. The output alignment network reverses this process for data traveling between the arithmetic processors and main memory.

In Fig. 5.15, note that the 4 × 4 matrix has two dummy columns stored, so it is stored as a 4 × 6 matrix. In this form, rows are accessed with a stride of 1, columns with a stride of 6, and diagonals with a stride of 7. Since 1, 6, and 7 are relatively prime to 5, in each case there are no memory conflicts when accessing the particular slice of the array of interest. If the matrix is stored without

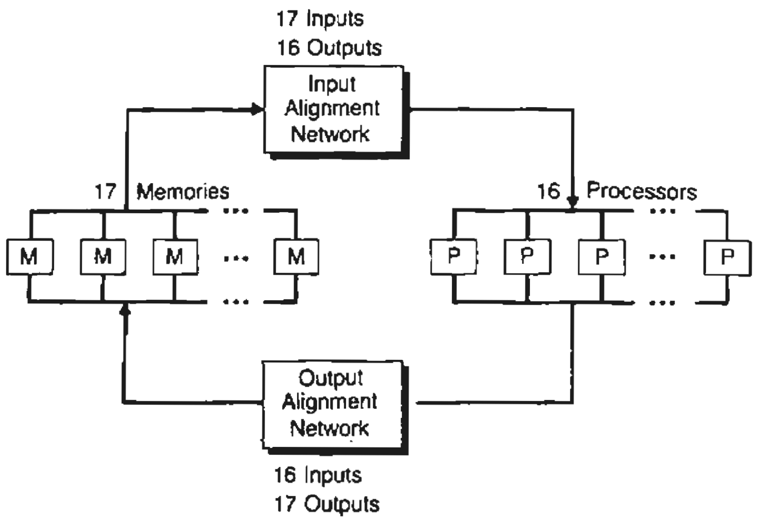

Fig. 5.14 The data flow and processor/memory structure of the Burroughs Scientific Processor.

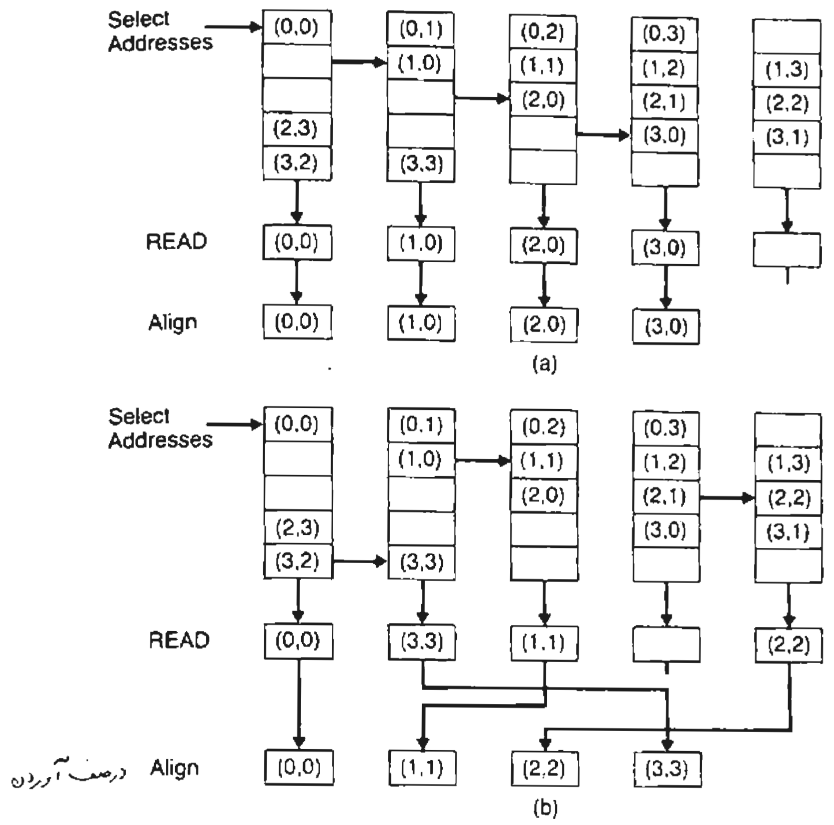

Fig. 5.15 A data structure that supports easy access to rows, columns, and diagonals: (a) Access to columns with stride 6; and (b) Access to diagonals with stride 7.

the dummy columns, then the stride to access diagonals is 5, which is equal to the number of memories and therefore causes a maximum number of conflicts.

The BSP processor was never sold and eventually the project was abandoned. Although the 17-memory structure solves some problems of access, it creates others. Addressing is more complex for this structure than for storage systems in which M is a power of 2. But more important is that the 17-memory system requires that access to the matrix components be made at the memory system, which is quite far from the processor. Obtaining a row of a matrix and then a column of the matrix, perhaps at a later time, forces the matrix to be in main memory and not in a buffer close to the processor. Hence, there is potentially high traffic to and from main memory just for the purpose of reformatting data.

Contrast the 17-memory structure with a Cray-like structure as shown in Fig. 5.10. The striking difference with respect to performance is that the Cray architecture drives the arithmetic units from a high-speed buffer memory (the vector registers), whereas the 17-memory structure drives the arithmetic units from a more remote main memory with two alignment networks contributing to storage delay. The high-speed buffer of the Cray provides for the possibility of loading data into the buffer just prior to when they are needed.

While data reside in the buffer memory, they can take part in multiple operations before being returned to main memory. Moreover, it is conceivable to provide a sufficiently large buffer memory so that reasonably large portions of matrices can be loaded into the buffer using an access pattern such as row access, that is supported by main memory.

The buffer memory can be structured for access to the various matrix components of interest, so once a matrix is loaded into the buffer, its elements can be accessed in any of several ways. A high-speed buffer can be structured to access the matrix by rows, columns, and diagonals by designing its cycle time to be equal to one clock cycle. For a one-cycle memory, the stride for pipeline access to a vector can be arbitrary.

The type of buffer we describe here is very costly when built in some popular high-speed technologies. A very simple alternative is to reformat matrices when necessary by transferring them between main memory and the high-speed buffer. For example, consider an 8×8 matrix stored by rows in an eight-module memory. If the next phase of the algorithm must access columns, we can reformat the rows from 8×8 to 8×9 by loading each eight-element row into the high-speed buffer and then storing back a nine-element replacement. The destination vector can be written to a different region of main memory to prevent overwriting of the source by destination during the reformatting. Since the row operations are pipelined, reading an entire row of eight elements takes only a little longer than reading a single element. After the matrix is restored to memory, it is in a format in which columns can be accessed with a stride of 9.

The reformatting time is approximately equal to the time required for two to four vector transfers, depending on the overhead per vector initiated and the startup time for a vector load or vector store. The reformatted matrix can be accessed by columns about d times faster than the original matrix, where d, as you recall, is the memory-access cycle time.

Depending on the value of d and the overhead per vector operation, the reformatting of the matrix may be the preferred way of gaining access to the entities needed. The reformatting process might well lead to less performance degradation than do the alignment networks shown in Fig. 5.15 because reformatting degrades performance only when it is needed, whereas the alignment networks tend to increase the latency of every vector fetch and store.

Architectures with high-speed buffers appear to have several advantages over architectures whose memory couples directly to an arithmetic unit. Although this observation is very dependent on existing technology, the trend today seems to be toward vector processors that use high-speed buffers to gain speed, as opposed to architectures that place needed operands far away from the processor that needs them.

The major design problem for the buffer architecture is building a memory that is both large enough to hold an interesting amount of data and fast enough to run at the clock cycle of the arithmetic units. The number of times that a datum in the buffer can be used in a computation before it is returned to main memory tends to decrease as buffer size decreases, so a small buffer may yield little or no savings in the total number of accesses to main memory.

Device technology has a strong influence on how designs will achieve variable-stride access in the future. Current trends suggest that the density of high-speed memory is increasing and that high-speed buffers, although very costly today, will tend to grow larger in the future. Cooling is another problem of importance because large amounts of high-speed memory packed very densely lead to potentially high power density per unit volume. The Cray II, for example, has so high a power density that it is cooled by immersion in liquid.

Technology trends suggest that both the power consumption per bit and the cost per bit are moving downward, which lends support to the evolution of high-speed buffers for variable-stride access as opposed to the BSP approach of handling variable-stride access exclusively in main memory.

5.4 Attached Vector-Processors

An important means for achieving economical high-speed computation is to provide for customization of each processor to the needs of each user. The idea is to partition an architecture into building blocks that can be combined in various ways to achieve different levels of performance with commensurate costs.

Figure 5.16 shows a basic high-speed conventional processor to which is connected a numerical processor that we call an *attached vector-processor.* The basic machine without the attached processor serves a large group of users with conventional workloads, and the machine with the attached processor satisfies the needs of the specialized group of users. This tends to reduce the cost to the specialized user because both the software and hardware of the general-purpose machine enjoy the advantages of the lower cost of high-volume production.

Some manufacturers of attached processors offer a model that can be connected to a variety of different host machines. Attached processors cover a very broad range of costs and performance, from low-cost units that attach to microcomputers to high-performance systems that attach to high-end commercial computers. Many commercial manufacturers offer vector attachments of their

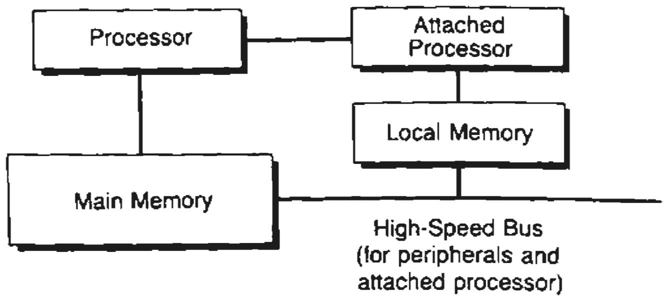

Fig. 5.16 The structure of a typical computer system with an attached processor.

own or a compatible model with a superset of instructions for vector operations. These approaches are used by Fujitsu, IBM, Hitachi, and NEC.

Our discussion in this section covers the generic architecture of an attached processor. We also give some specific details regarding the FPS-164 from Floating-Point Systems by way of example to make the details more concrete. Charlesworth and Gustafson [1986] provide interesting background information on this topic.

We know from prior discussions that vector access to rows and columns, and possibly to other matrix components, are essential for efficient numerical computations. This requirement forces the architect to design the memory system to support such access, but places very few constraints on the design of the arithmetic processor. The arithmetic unit should also be structured to support the most common and demanding needs of the users. So let us review a few of the algorithms encountered earlier in the text.

For most numerical applications, the solution of linear equations of various forms is the most central requirement. Linear programming requires related techniques to solve constrained optimization problems. Even for nonlinear problems, linear techniques are very important.

Nonlinear systems of equations are often solved by iterative linear methods. The idea is that some nonlinear systems behave linearly with respect to small perturbations about a solution. Consequently, it is possible to produce a full trajectory for a nonlinear solution from a sequence of solutions to a linear system that describes the small-perturbation behavior of the nonlinear system. Iterative techniques are often employed to produce a solution to the full nonlinear system from the solution obtained by using the linear approximation.

For both linear programming and linear algebra operations, the inner loop of the computation often takes the general form

$$a := a + b \times c \tag{5.1}$$

where a, b, and c are scalar. In a general-purpose structure, the product can be computed and stored in a register and then added to a sum stored in a different register.

Since this operation is so common, we can make it a three-operand operation and provide for both the multiplication and addition to be done in one arithmetic unit, without requiring an intervening store and load of the product to and from a high-speed register. The structure commonly used takes the form shown in Fig. 5.17, in which two operands enter a multiplier whose output is tied directly to an adder, to which a third operand is connected.

Equation (5.1) can be evaluated in several different contexts, depending on the order in which data are presented to the arithmetic unit. The most efficient computation occurs when Eq. (5.1) is used to produce a vector of outputs from a vector of inputs. Using our vector notation, Eq. (5.1) in this context becomes

$$a[1 \ldots N] := a[1 \ldots N] + b \times c[1 \ldots N] \tag{5.2}$$

An efficient pipeline implementation of this equation provides for loading a scalar variable to one input of the multiplier and streaming vectors **A** and **C** through the arithmetic unit. The output vector is the updated **A** vector, which is returned to the buffer storage area reserved for **A**.

Another possible context for Eq. (5.1) is one in which two vectors are reduced to a scalar by an inner-product operation, which produces a single scalar output from two vector inputs. This form of Eq. (5.1) is

$$a := a + b[i] \times c[i] \tag{5.3}$$

where the products of the form $b[i] \times c[i]$ are accumulated into the scalar variable a. The initial value of a is zero when an ordinary inner product is required. However, some algorithms use Eq. (5.3) in a manner that requires a nonzero initial value for a.

The difficulty with Eq. (5.3) is that there is an interlock required between successive iterations since the output variable a for one iteration is an input variable for the next iteration. If an addition is performed in a pipeline with d units of delay, the interlock may require as many as $d - 1$ idle times between

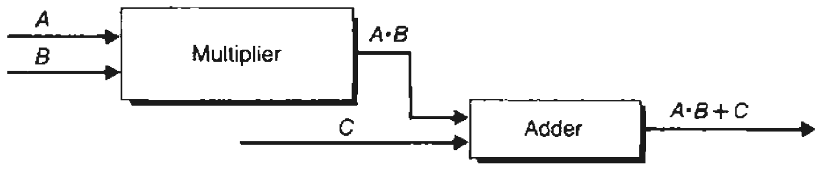

Fig. 5.17 The structure of a multiply-adder.

successive outputs in order to give the pipeline time to compute a new value for variable a to be used in the next iteration. This is as much as d times longer than the execution time required for Eq. (5.2), and the inefficiency arises only because of the interlock used.

A way around this problem is described in Kogge [1981] and is discussed in Section 3.4.5. The trick is to produce d different sums by computing Eq. (5.3) according to the schedule

$$a_i := a_{i-d} + b_i \times c_i \tag{5.4}$$

The subscripts in Eq. (5.4) denote the operand that appears at the arithmetic unit input or output at time i. This form of the computation does not require any interlocks because a_{i-d} is available for use at a pipeline input just after it emerges from the output end of the pipeline.

Unfortunately, Eq. (5.4) produces d distinct sums, which is not the intended result of Eq. (5.3). So at the completion of the calculation described by Eq. (5.4), it is necessary to sum the d output variables into a final result. The final summation requires a small additional time that degrades performance negligibly when the **B** and **C** vectors are long. The performance degradation cannot be neglected when the **B** and **C** vectors are short, in which case the methodology described by Eq. (5.4) should be avoided in favor of an alternate problem formulation that makes more efficient use of the architecture.

The FPS-164 processor [Charlesworth and Gustafson 1986] is an example of an attached processor. Figure 5.18 shows that the vector processor has its own main memory, high-speed scalar arithmetic, and a variable number of pipelined

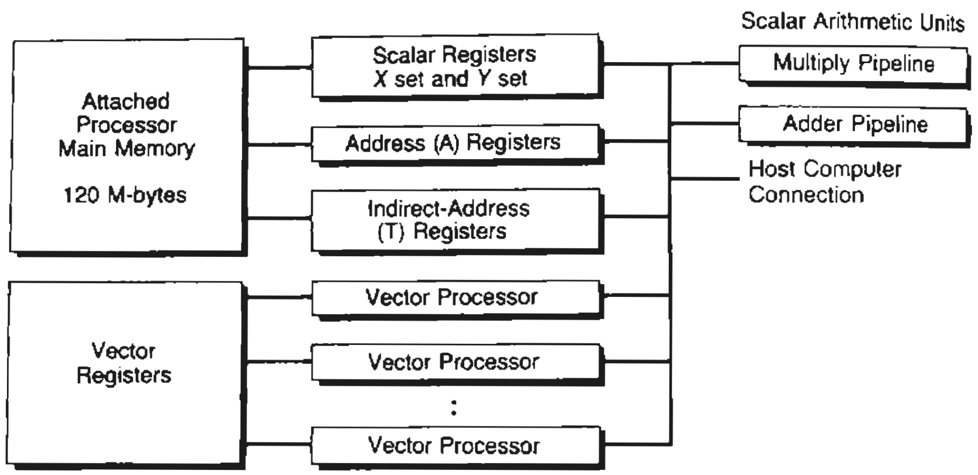

Fig. 5.18 The structure of the FPS-164 attached processor.

vector units. The system has a high-speed connection to a host computer. The host function is to provide data and programs for the vector processor and to receive results when they are available. The vector processor is designed specifically for high-speed floating-point operations and has virtually no support for general applications and utility functions. These are supported by the host.

Note that the scalar processor shown in Fig. 5.18 is built for fast scalar operations in that it has a separate multiplier and adder, two sets of operand registers (*X* and *Y* registers), one set of address registers (*A* registers), and a set of indirect-address registers (*T* registers). The scalar processor broadcasts instructions and data to up to 15 vector processors, one of which is shown in block-diagram form in Fig. 5.19.

The vector processor has two multiply-add units, each capable of producing one output per cycle. There are two sets of vector and scalar registers and an input that receives data broadcast from the scalar processor. To make the best use of the vector processor, this architecture is designed to have sufficient buffer space locally in the vector processor to eliminate some loads and stores of vector data. Consequently, the vector registers are very long, 2K operands long, and there are four vector registers in each of two sets of registers. Thus one processor can hold 2 × 4 × 2K = 16K elements from vectors.

The scalar registers are far less numerous. Each of two sets holds four operands. The reason for having four scalar operands is that, for any given

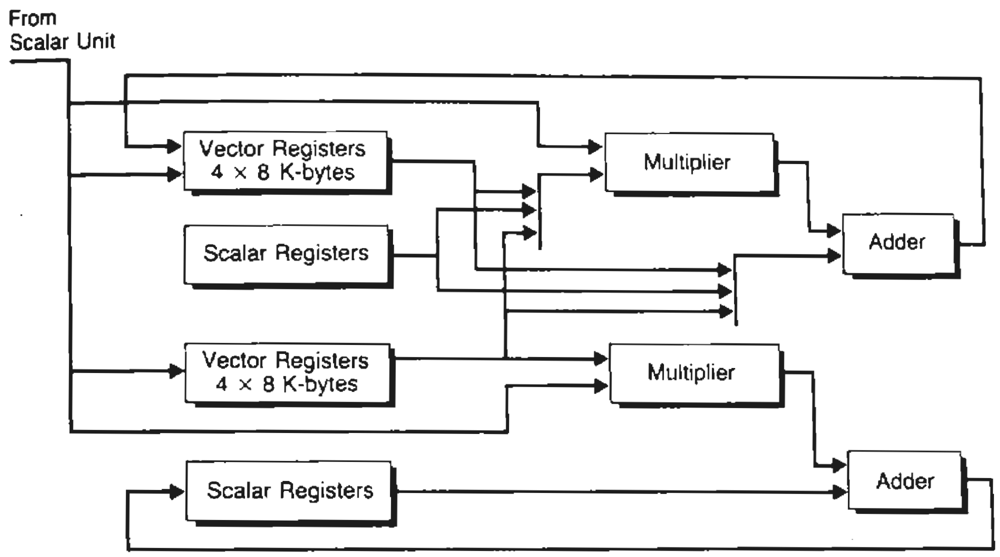

Fig. 5.19 The structure of an FPS-164 vector processor.

vector, up to four different scalar multiples of that vector can be computed without the need to obtain new data. This tends to reduce traffic to memory in that each vector can be used up to four times once it is loaded into a vector processor, and therefore it is not necessary to store and reload the operand vector. So there has been a deliberate effort in this design to design the number of scalar registers and the size of the vector registers in such a way as to reduce memory traffic.

The operation of this processor is rather interesting. The vector processors act as slaves to the scalar processor. They receive instructions and data from the scalar processor—individually or in a broadcast mode that transmits data or instructions to all vector processors simultaneously. In this mode the scalar processor can also read selectively from the registers of any selected vector processor.

The normal mode of operation is to load individual vector registers with starting data, with this done selectively rather than in broadcast mode. Thereafter, scalar data and instructions are broadcast, and the processors react synchronously, each performing the same step, but operating on different data.

When the scalar processor transmits in selective mode rather than in broadcast mode, all processors except the receiving processor are idle. Therefore this mode is used as infrequently as possible. Since the vector registers can hold collectively as many as $15 \times 8 \times 2000 = 240{,}000$ operands, two or more matrices of rather substantial size can be stored within the vector processors. This tends to reduce the need to store and reload data selectively to and from the vector registers.

Vector operations can be performed concurrently with scalar operations that take place in the scalar processor. Hence the architecture provides for overlapping the serial computations that constitute loop overhead with the parallel execution of the prior loop. Earlier in this text, this process has been described as an essential aspect of efficient processing.

The machine is heavily oriented to typical computations associated with large-scale numerical processing. The benefit of using this approach is that its users can purchase only what they need, since they can purchase as many or as few vector processors as they can justify. Moreover, they can use an existing on-site processor as a host and need not support the design and development of a distinct host.

We discuss the role of the indirect-address pointers in the next section, which focuses on techniques for handling sparse matrices.

5.5 Sparse-Matrix Techniques

In many matrix problems, relatively few elements of a matrix are nonzero. Such matrices arise in finite-element problems in which a nonzero entry represents the interaction of one element of volume with a neighboring volume element.

The number of nonzero elements is related to the number of neighbors per volume element and is generally a very small fraction of the total number of matrix entries.

These matrices are very similar to matrices that describe continuum-model problems, and indeed they should be, because the finite-element model is a continuum model. The difference is the irregularity of the surface or volume that is being modeled. In modeling the stresses on an airframe, for example, near-neighbor descriptions of a cylindrical fuselage produce a sparse matrix whose structure leads to very simple near-neighbor operations. If the model extends beyond the fuselage, however, the problem can become very difficult to solve. If the model includes the wings, for example, then, at the place the wings are joined to the fuselage, we must include some interactions that explain the stresses likely to be found there. These interactions give rise to nonzero elements that lie in the matrix in relatively unpredictable places.

When a sparse matrix is highly structured with no irregularities, it is often possible to deal with the nonzero elements exclusively. In the continuum-model problems investigated earlier, this is precisely what the programs do. In two dimensions, a typical code accesses the four nearest neighbors, and no other accesses are required.

If we move to a finite-element description of an airframe, then near-neighbor accesses suffice for most interactions, but the remaining interactions, such as the ones that describe the stresses where the wing joins the fuselage, require nonlocal accesses. Moreover, the nonlocal accesses need not follow any uniform or predictable pattern. Hence, to process only the nonzero matrix elements may require rather rich and expensive interconnections. Moreover, the interconnections may need to be used selectively rather than in parallel because of the absence of regularity in the distribution of the nonzero elements in the sparse matrix.

Several approaches have been used in architectures to solve sparse-matrix problems. An early attempt in the CDC STAR created what was known as *sparse vectors*. A sparse vector consists of two vectors—one is a short vector that contains just the nonzero elements of a vector, and the other is a bit vector whose 1s indicate where the nonzero elements belong, and whose 0s represent the zeros in the vector. The length of the bit vector is equal to the length of the sparse vector, but there is a 64-to-1 reduction in the number of bits when the vector elements are 64-bit operands.

When accessing or storing sparse vectors, the CDC STAR uses the bit vector to determine whether an access has to be made in a particular index position. The access is skipped if the bit vector has a 0 in the corresponding position. Although the bit vector can reduce the number of memory accesses, the items that are accessed may lie in conflicting memory modules, which leads to delays in the pipeline. This can negate some of the performance gain attributed to the accesses saved by dealing only with the nonzero elements. There is a small additional processing overhead per 0 in the bit vector, but not as large a penalty as a full memory access.

Obviously an architecture of this type can incorporate various other facilities for sparse vectors, such as the ability to translate a vector from sparse format into a full vector format and to translate back again. Also, the pipeline arithmetic units can be organized to accept sparse vectors at their inputs and to produce sparse vectors at their outputs by doing conversions on the fly from sparse to computational and back to sparse.

The major problem with this approach is that there is only a 64-to-1 reduction in the information saved since, at best, it still takes a single bit to represent 64 bits. Large sparse matrices are so sparse in many applications that a 64-to-1 improvement is minuscule compared to what is possible. How this basic approach might be extended is still an open question for research.

An alternative method for representing sparse matrices is to store only the nonzero elements, and with each array of elements store a list of indices in the original matrix. It may be necessary as well to invert this structure by mapping indices to pointers by a hashing scheme that maintains a compact storage representation of the inverted list.

If the hash lookup finds an index, then the corresponding element is nonzero, and the hash table contains the storage address of the corresponding datum. If the hash lookup fails to find an item, the corresponding item has a zero value. Hashing for access to data is very much like a cache lookup. Just as a cache lookup can be pipelined, so can hash access, and therefore this method for dealing with sparse arrays is potentially useful in pipeline computers.

Returning to Fig. 5.18, the *T*-registers in the scalar processor contain the indices of nonzero elements of a sparse matrix. When operations need to be done for nonzero items only, as each new item is accessed, the scalar processor finds the address of the next nonzero element and fetches that datum instead of fetching the next sequential datum. The program has to deal with the zero elements that have been skipped, but the cost of skipping and the additional performance degradation from memory contention can be very small relative to the large gains in processing speed due to the elimination of processing the 0s in the sparse matrix.

Apart from methods related to the representation of sparse matrices are algorithms that perform computations only on nonzero elements of sparse matrices. The major difficulty is to develop such algorithms when the sparse matrix does not have a simple structure. Hoshino [1989] gives an example of how to treat a sparse problem that is almost tridiagonal, and is an excellent model of a successful methodology that can be used for sparse problems in which the majority of the nonzero elements fall into a particular structure. The problem treated by Hoshino is the solution of a block tridiagonal system of linear equations. The nonzero elements of the **A** matrix lie exclusively in smaller blocks that lie on the diagonal or immediately above or below it. The small blocks are themselves tridiagonal matrices. Standard techniques for tridiagonal matrices can reduce this system to a smaller system of equations, but the reduction cannot be taken to the full solution of the equations when the original equations are block tridiagonal. However, the result of the reduction produces a reduced set of equations that is solvable, in this case by standard tridiagonal techniques. For finite-element codes, in particular, sparse techniques reduce the equations to a much smaller set that may well be dense or have a sparse structure that can be exploited. Consequently, there is hope that sparse problems that arise in actual practice can be solved successfully on a parallel computer with high efficiency, but as yet this problem area has not been deeply explored.

This completes our discussion of sparse-matrix techniques. In the next section we take a quick tour of a very high-performance machine somewhat different from the ones mentioned thus far in the chapter.

5.6 The GF-11—A Very High-Speed Vector Processor

This chapter has assumed that pipelining techniques are the principle techniques for vector processors. The FPS-164 example suggests that pipeline processing may not give enough performance, and that some combination between pipeline and fully parallel implementation may be useful as well.

In this section we describe a machine architecture developed by IBM that is yet another combination of pipelined and parallel design, with a much stronger parallel component than the FPS-164 has. The machine is called the GF-11 [Beetem, Denneau, and Weingarten 1985], which stems from its peak performance of 11 Gflops (11000 Mflops).

The general structure of the GF-11 is very much like a richly connected ILLIAC IV; it appears in Fig. 5.20. The interconnection network is capable of producing any permutation whatsoever among the 576 processors in the system. The interconnection network is a three-stage network with two shuffles between the three stages. However, these shuffles lie between 24 × 24 crossbar switches, as shown in Fig. 5.21, rather than between 2 × 2 switching elements. This network is sometimes called a *Benes* network [Benes 1964], and it is known to be capable of producing an arbitrary permutation.

Since the GF-11 is a vector processor, it issues vector instructions from a control unit, and they are obeyed by the 576 processors. The memory per processor is modest—64 K-bytes of high-speed and 256 K-bytes of slower-speed memory—but the total memory in the GF-11 is very large because of the multiplier of 576. The slow memory alone accounts for 144 M-bytes. Slow memory is expandable to 2M per processor as higher density chips become available, which allows expansion to 1.152 G-bytes in the system.

The processor speed is several times faster than the speed of fast local

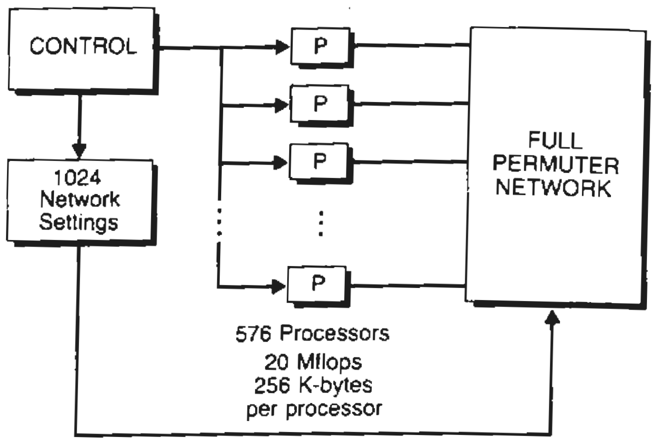

Fig. 5.20 The structure of the GF-11 research machine.

memory. Consequently, each processor has a very high-speed register file that serves as the fastest level of the memory hierarchy. The arithmetic processor itself is pipelined to maintain high throughput for floating-point operations. Hence the pipelining occurs mainly within the arithmetic unit, and the high-replication factor of 576 gives the extraordinary throughput for the system.

The primary purpose for the construction of this processor is to solve a problem in quantum chromodynamics whose solution can produce the mass of various elementary particles through lengthy calculations. If the computed mass is equal to the observed measurements of mass, the predictions of the underlying theory will be confirmed, thereby lending some evidence that the theory is correct. If not, the theory needs to be modified or abandoned. Unfortunately, the computation involves the evaluation of very slowly converging multiple integrals. At the rate of 11 Gflops, the computation takes about one calendar year.

The structure of the GF-11 is vector-oriented, with a single broadcast instruction stream. This structure is used because the quantum chromodynamics problem calls for repeated summations that must be synchronized across all processors. The communication requirements of the problem stem from reliability considerations. The GF-11 programs are designed for only 512 processors, and the idea is to use the 64 remaining processors as spares. Should any processor fail, it can be quickly mapped out of the array, and a spare processor can be mapped into the array in its place. The machine then needs to be restarted from the last checkpoint, but it should continue to operate at full speed after it is restarted.

The switch permutation is controlled by a collection of bit vectors stored in

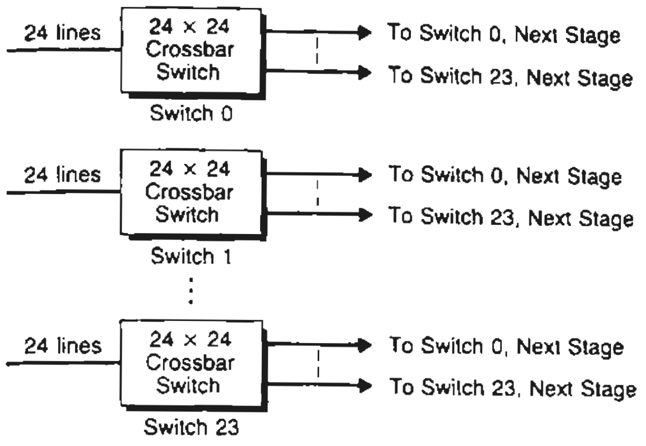

Fig. 5.21 Detailed view of a portion of the GF-11 full permuter switch. The 576 lines pass through three ranks of switches, one rank of which is shown here. Each switch is connected to all 24 switches in the next rank.

the memory called *permutation memory* in Fig. 5.20. This memory holds 1024 bit vectors, each selectable by a 10-bit index issued from the control unit. To perform a specific permutation, the controller issues the 10-bit index to the permutation memory. Then the bit vector produced by this read is loaded into the switch, and the settings are made. Then data traverses the switch.

The quantum chromodynamics problem uses only 6 of the 1024 possible settings. In the event of a processor failure, it is relatively straightforward to compute a new bit vector that maps out the failed processor and replaces it with a spare. When that bit vector is stored in the permutation memory, the computation can proceed.

The GF-11 is a research vehicle, not a commercial machine. If it is successful as a research machine for solving the problem posed, that does not mean that this architecture is cost-effective for problems in general. However, its rich interconnection structure enhances the GF-11's ability to execute more problems in general. The major constraint on GF-11 programs is that all processors execute the same instruction stream.

5.7 Final Comments on Vector Computers

It is interesting to contrast and compare the ideas that emerge from the discussion in this chapter to see their strengths and to identify future trends in vector machines. We have explored the pure pipeline structures of the CDC STAR and Cray I, the combination of pipeline and parallel structure of the FPS-164 and the GF-11, and in Chapter 4, the purely parallel structure of the ILLIAC IV. These machines span a rather broad set of design choices. The major trends identified are to:

- Provide vector instructions to take advantage of this approach for numerical applications;
- Provide facilities to extend the range of applicability of the architecture beyond vector processing;
- Use multiple levels of memory, particularly high-speed buffers; and
- Mix pipeline and parallel techniques in various degrees to achieve an acceptable value of price to performance.

On the other hand, the characteristics that differ from processor to processor concern the specific design choices that trade-off speed against cost and flexibility against efficiency. No one design is best. Choices were driven in many cases by available technology, which differed considerably for the designs described in this text. Had all designers been given the same underlying technology, some individual design choices might be common among several architectures, but even then it is unlikely that any two designs would be markedly similar.

The examples we discuss show the final choices of the designers, and our discussion illustrates various aspects of the choices that affect cost and performance. Unfortunately, we are not able to present the interesting choices that were investigated along the way and abandoned for various reasons.

The trends listed here are by no means the only ones that exist, nor can we rule out new trends in the future as technology makes new designs possible. The future architect should use this discussion as a guide, but not as an exhaustive treatment of the subject. Examine any attractive idea and be prepared to develop it into a full-fledged machine design. But be sure to examine it carefully. Rarely is there a case for building a machine that is handicapped by some inherent inefficiency.

The major implementation technique for vector machines appears to be pipelining. We see two basic reasons for this to be true:

1. Pipelining provides a means for coupling a slow memory to a fast arithmetic unit.
2. Pipelining enables arithmetic units to produce a sequence of results at a rate much faster than their inherent latency in forming a single result.

If we view the memory system and the arithmetic system as two distinct bottlenecks in a conventional computer system, then we can see that pipelined architecture attempts to relieve both bottlenecks. For memory systems, the rate at which operands are produced at a memory port is anywhere from 5 to 20 times the rate at which the memory system cycles one memory module. Similarly, the rate at which sums and products are produced at the output of an arithmetic unit is from three to ten times faster for a pipelined structure than for a conventional serial structure. These are significant speed improvements whose individual cost is relatively low compared to the cost of a full computer system. Consequently, we can expect to see the continuation of the trend to build pipelined arithmetic units driven by pipelined memory systems.

In the last chapter we introduced six technology constraints that have to be overcome by a high-performance design. In this section we review those constraints and discuss how a vector architecture deals with them in achieving high performance.

- *Processor bandwidth:* Two major ways of boosting processing bandwidth are discussed here. Pipelined arithmetic is probably the most widely used method because of its high performance at relatively low cost. To deliver speeds beyond those available from pipelined arithmetic alone, replication of arithmetic units into fully parallel systems is the technique of choice.
- *Memory bandwidth:* For main memory, designers build large memory systems from multiple independent memory modules. Bandwidth grows with the number of independent modules. To match the bandwidth of arithmetic units, one or two levels of high-speed memory are used, most frequently in the form of addressable registers. The trend is to make the high-speed buffers very large, offering from 256K to 1M of storage currently and larger storage in the years to come.
- *Input/output bandwidth:* Although we have not discussed input/output in this chapter, it is clear that input/output bandwidth can be increased proportionally to increases in memory bandwidth if we assume that input/output operations require a fixed fraction of memory operations. Most high-performance systems incorporate 10 to 20 direct memory-access channels whose speeds are compatible with main memory speed. They rely heavily on a high-memory bandwidth, usually obtained through the use of multiple-memory modules.
- *Communication bandwidth:* the majority of the vector architectures discussed in this chapter do not require processor-to-processor communication. Information is distributed among operands within the vector operation itself by means, for example, of a scalar product that combines information from all elements in two vectors. Information is also distributed among different vectors through a common memory. The arithmetic unit can obtain operand values that are the results of various computations by accessing those values in main memory. In this case communication bandwidth and memory bandwidth refer to the same quantity.

  The exception in this chapter is the GF-11. It has local memory only, and computations interact through an interconnection network. This bandwidth is made very high by disallowing conflicts in the network. The interconnections are set according to precomputed control data, so they yield a useful processor-to-processor permutation. The communication bandwidth is comparable to memory bandwidth.

- *Synchronization:* for one pipeline, synchronization is accomplished automatically because operations are performed in the order in which they enter the pipeline. For multiple pipelines, the FPS-164 approach is to synchronize by using a single program to control all pipelines. The GF-11 approach is similar in that a single control unit issues a broadcast command to all processors. Both the FPS-164 and GF-11 synchronize all processors at each step through the instruction stream. The Cray architecture uses pipeline interlocks to control vector operations so that nonconflicting operations can be done in parallel, and dependent operations are chained to overlap as much as possible, provided that the overlap does not create an incorrect answer.
- *Multiple purpose:* the vector machines discussed in this chapter tend to be useful over a large class of vector problems, mainly because they support a variety of data-access modes and have rich sets of vector instructions. Nevertheless, their utility is biased strongly to numerical applications, and it is not clear that they are efficient for nonnumerical applications.

These characteristics clearly show that the major advantages of a vector architecture are:

- Efficient use of memory bandwidth through pipelined access;
- The excellent cost-performance of pipelined arithmetic; and
- The very simple mechanisms that serve the needs of communication and synchronization.

These three characteristics yield a combination of high performance and high efficiency. Unfortunately, they do not yield a system that is well suited to all purposes. In general, vector processors have a much larger area of applications than do continuum-model processors with near-neighbor connections. The key difference is that vector processors can deal with both local and remote operands by making use of a large random-access main memory to fetch data at arbitrary locations. Processors with only near-neighbor mesh connections have limited ability to reach remote data. With a larger realm of application, vector processors have created an important niche in computing and are far more widely used than continuum-model processors.

Exercises

5.1 The purpose of this exercise is to explore techniques for implementing the individual operations of Program 5.1 (in Section 5.2.1). The algorithm scans a column for a maximum element, pivots by interchanging rows, then updates a partial row, partial column, and a square subarray. Your objective is to work out a pipelined architecture that can perform each of the processes of scan for maximum, pivot, row update, column update, and subarray update individually in pipelined fashion without requiring overlap between operations. Your goal is to produce one result per machine cycle within each process, but you are allowed to have periods between processes during which no results are produced. (For the MAX operation, try to produce one comparison per cycle.)

For timing, assume the following total delays per operation:

Memory access	four cycles
Add	two cycles
Multiply	two cycles
Divide	eight cycles

You are constrained to use main memory for vector storage since your arithmetic subsystem has insufficient register storage to hold vectors or substantial portions of vectors. In your answer you must show at least:

- The storage format of the array in memory;
- The machine instructions for each of the processes (with a clear description of the action of these instructions);
- A block diagram of the computer system showing the principal elements; and
- A discussion of the way that each of the five processes is handled.

5.2 The object of this exercise is to write programs for a processor designed for vector operations. Carefully study Program 5.1 in Section 5.2. Assume that it is executed on a processor similar to the one shown in Fig. 5.1. There are 64 independent memories, and the matrix is 32 × 32. Memory operations take 12 machine cycles, and all arithmetic results are delivered by vector instructions at the rate of one per cycle.

a) Consider operations on a matrix stored with entire columns in individual memories and rows stored across memories. For each major portion of the program, what speedup is achieved?

b) Consider the matrix stored in skewed format so that rows and columns are each accessible in a single access. What is the speedup for each major section of the code? Consider the effects of nonunit stride when calculating the speedup.

c) If variable delays are used to align vectors, what is the maximum length of each delay required for any vector allocation, assuming that vector access for both sources and the destination have a stride of 1?

5.3 The purpose of this exercise is to contrast the results obtained for a pipelined architecture in the previous problem with a parallel vector architecture similar to the ILLIAC IV. For this problem assume that the 32 × 32 matrix problem of Exercise 5.2 is to be solved on an ILLIAC IV architecture that contains 64 processors connected as an 8 × 8 array. Each processor is connected to four processors whose indices differ by +1, −1, +8, and −8 modulo 64.

a) Let the matrix be stored with columns in individual memories and rows across memories. For each major portion of the program, what speedup is achieved?

b) If the matrix is stored in skewed format so that rows and columns are each accessible in a single access, what is the speedup for each major section of the code? Assume that deskewing can be done at no cost.

c) If each unit shift requires one cycle, what is the speedup for each major section of code in this version of the program?

d) If you use the interconnections as given to speed up the deskewing, and each single interconnection takes one cycle, what is the speedup for each major section of code in this version of the program?

5.4 The algorithm for Gaussian elimination used in Exercise 5.3 accesses both rows and columns of a matrix. Access in two different dimensions may reduce efficiency, and it is worthwhile to modify the algorithm to work out a variation that uses column access only.

a) Consider how to change the algorithm so that it accesses columns only, yet is faithful to the intent of the original algorithm and has an efficiency that is competitive with (but possibly poorer than) the algorithm implementation of Exercise 5.3. Describe your new algorithm briefly, then give a detailed discussion of the portions of the algorithm that differ from the implementation in Exercise 5.3.

b) The new portions of the algorithm may require somewhat different architecture than that described in Exercise 5.3. Describe an architectural design that is well suited to implementing those portions of the new version of the algorithm that differ from the corresponding portions of the old version. Give enough information to establish that your implementation is efficient and reasonable.

5.5 The purpose of this exercise is to examine vector pipeline techniques. Consider the recurrence equation, Eq. (4.13), and explore its implementation in a pipelined computer system.

a) To obtain maximum parallelism, Eq. (4.13) should be solved by recursive doubling. Find a recursive doubling solution or use the solution obtained in the answer to Exercise 4.7.

b) Show the block diagram of a specialized pipeline to evaluate one cycle of a recursive doubling version of the recurrence. This diagram can be broken into blocks that are addition, subtraction, multiplication, and division. The blocks are assumed to be multicycle floating-point units, each capable of being pipelined with a rate of completion of one result per cycle.

c) Show how to stream the constant vectors into a processing unit based on your pipelined design so that the recursive doubling solution is fully pipelined. Use delays in place of interlocks and attempt to produce results at the rate of one result per cycle. Use multiple copies of the units designed in *b* and give the structure of the full processor by showing how to connect each of the copies from *b* with extra delays to produce the answers to the recurrence.

5.6 The purpose of this exercise is to contrast caches with high-speed storage registers in systems that use vector arithmetic. Reconsider the Gaussian elimination algorithm of Program 5.1, operating on a 64×64 matrix. In this exercise, when we refer to the array-update process, we refer to the innermost loop of that algorithm.

a) Assume that there are 32 independent memories, each capable of a two-cycle access time. Compare the relative timing for accessing a row versus a column when accesses are pipelined.

b) Now consider the effect of a cache. The cache consists of 64 sets, two-way associative, with each line of the cache holding one operand. Assume that a single vector of length 64 is accessed two consecutive times and no intervening access is made. For the second vector access, how many misses will there be if the vector is a row vector? How many if the vector is a column vector? State your assumptions on the storage format and the mapping of addresses to cache lines.

c) In one iteration of the array-update process in the algorithm, how many misses will there be? (To simplify the calculation, you may assume that all vector accesses are of length 64, even though the actual length depends on the specific iteration of the algorithm.) Note that you may completely ignore the other accesses as if they were not present at all. A miss is defined to be an access to an item that is not present because it either was not accessed in the previous iteration of the algorithm or was displaced from the cache because at least two other lines in the same set were accessed more recently.

d) Now assume that there are two vector registers, each 64 items long. Show how to load data into the vector registers to reduce the number of data accesses to memory as much as you can.

e) Use the data you have developed and parameters given to calculate the relative number of accesses to main memory for the cache-based computer and the register-based computer when the array-update process is performed.

5.7 The inner loop of an algorithm performs the following operation:

$$Sum := Sum + b \times C[i]/d$$

Assume that b, $C[i]$, and d are variables that can be streamed into a pipeline from memory with one cycle access to each variable, so that memory is not a bottleneck for computation. The objective is to perform the operation given to produce the final sum in minimum time.

a) Design the block diagram and functional behavior of a three-function pipeline whose operations are multiply, add, and divide. Find the collision vectors for controlling the system and the fastest possible cycle for the sequence of multiplication, division, and addition when operating on independent operands. (This does not account for the interlocking necessary to make sure that the value of *Sum* used as an input is derived from the most recent value of *Sum* as an output.)

b) Now consider the maximum speed attainable when the input to the adder is interlocked to the output of the adder. What is this maximum speed in your design?

c) If we want to produce one update of *Sum* per cycle on the average, how can we structure the computation to come close to achieving this rate?

5.8 Consider an architecture similar to the Burroughs' Scientific Processor (BSP) in which 17 items are read from memory, but only 16 are delivered to the arithmetic unit.

a) For a 16 × 16 matrix, consider how to select the elements of a column and permute them into column order for delivery to the arithmetic unit. What are the selection and permutation operations required to access Column 0? Column 3? Column 4? Assume that the matrix rows are stored across the memories.

b) What are the selection and permutation operations required to access Row 0? Row 3? Row 4?

c) What basic permutations would you build into this machine to facilitate row access? What permutations would you build in to facilitate column access?

5.9 Refer to Fig. 4.20 for a visual description of the data flow required for a bitonic sort. Construct a vector algorithm that does the bitonic sort of N numbers in O(log N^2) vector operations. Assume that you have a vector operator that reads the odd-numbered indices of a vector of length N and compresses them into a dense vector of length $N/2$. Use this vector operator to create an inverse perfect shuffle. Construct the bitonic sorter from inverse perfect shuffles.

5.10 a) Exercise 5.9 describes how to create vector operations that can implement the inverse perfect shuffle. Describe a vector implementation of an instruction that operates on two vectors, each of length $N/2$ and creates a vector of length N that is the perfect shuffle of those vectors.

b) Consider the implementation of the perfect shuffle as described in part *a* on a computer such as that described in Fig. 5.7 and whose timing diagram is similar to that shown in Figs. 5.4 and 5.6. Assume that **C** (of length 16) is the perfect shuffle of the vectors **A** and **B**, each of length 8. All three vectors have their first element in Memory 0. Assume that the memory takes two clocks per access and that there are 8 memories. Construct a timing diagram similar to Fig. 5.6 that describes the shuffle operation applied to **A** and **B**.

c) Construct a timing diagram that shows the behavior of the processor on vectors **A** and **B** of length 16.

d) Your pipeline may suffer extra contention after initial startup. How many cycles do you lose in part *c* due to this contention?

5.11 Figure 4.16 shows the structure required to compute an FFT of eight data points. Use the perfect-shuffle vector instruction from 5.10 in a program to compute the FFT of N data points. Assume that the operations at each node are "black box" computations whose internals are unknown to you and are unimportant. The objective is to show how you achieve the data flow required for the FFT by means of the instructions available.

5.12 The reverse binary vector operation takes the element at index i in a vector whose length is a power of two, and moves the element to the position whose index in binary is the reverse of the binary representation of i. Show how to implement a reverse-binary operation by vector operations that select components of a vector and write these components in their same relative position in another part of storage. The selection is made by means a bit vector whose 1s denotes which components to copy and whose 0s designate a component that is not copied. Assume that the length of the vector is a power of 2, and show that the number of vector operations required is proportional to the logarithm of the vector length.

6

Sat cit si sat bene. [It is done quickly enough if it is done well.]
—*Latin proverb*

Multiprocessors

Thus far we have treated methods for speeding up a single instruction stream. Although there is but a single program in execution, the designs discussed earlier exploit concurrency within the instruction stream and within individual instructions. In this chapter we turn to the discussion of *multiprocessors*—computer systems composed of several independent processors. The motivation for moving toward multiple processors is strictly a matter of performance because device technology places an upper bound on the speed of any single processor. To exceed that bound requires multiple processors.

The central themes of this chapter are multiprocessor structures and performance. Our objective is to show several interesting techniques for organizing multiple processors into highly parallel systems and to give insight into the potential performance improvements and bottlenecks of such systems. Chapter 7 treats software strategies for using the available parallelism of these systems.

6.1 Background

Our earlier discussions of high-performance machines study two important classes of parallelism. Pipeline machines produce high performance by placing several stages of a pipeline in operation simultaneously. Machines for continuum calculations have multiple processors, each executing the same program. In both cases, a single program is used to operate on vectors or arrays of data. Flynn [1966] termed this type of parallelism *single-instruction stream, multiple-data stream* (*SIMD*) parallelism. Recall, for example, an extreme implementation of this idea in the form of the GF-11, in which each of 576 processors executes identical instructions broadcast to them by a single control unit.

Another SIMD machine with massive parallelism is the Connection Machine [Hillis 1986] with 64K 1-bit processors. The architect is truly fortunate when an application can be executed on machines that are built around the lock-step parallelism required for SIMD machines because the architecture efficiently executes programs well suited to SIMD execution.

High performance on such machines requires rewriting conventional algorithms to manipulate many data simultaneously by means of instructions broadcast to all processors. Although programming for these machines can be difficult in principle, in the ideal case, a serial algorithm can be converted to an SIMD algorithm by replacing each inner loop with a single broadcast instruction that implements the complete loop. The fact that an important, but limited, class of problems fits this model extremely well has provided the impetus for the design and construction of these machines.

Clearly, some large problems do not lend themselves to efficient execution in an SIMD architecture. The operations required for such problems cannot easily be organized into repetitive operations on uniformly structured data. They tend to be unstructured and unpredictable. Addressing patterns tend to be data dependent, so the architecture cannot easily preload data by anticipating future accesses.

The architect who must attain high performance for such problems inevitably looks for a solution in a multiprocessor structure. Such an architecture is composed of several independent computers, each capable of executing its own program. Flynn [1966] calls this type of architecture *multiple-instruction stream, multiple-data stream* (*MIMD*) architecture. The processors of a multiprocessor are interconnected in some fashion to permit programs to exchange data and synchronize activities.

A model of such an architecture is shown in Fig. 6.1. In this figure each processor has registers, arithmetic and logic units, and access to memory and input/output modules. In Fig. 6.1(a) we show the memory and input/output systems as separate subsystems shared among all of the processors. Figure 6.1(b) shows the memory and input/output units attached to individual processors. No sharing of memory and input/output is permitted in Fig. 6.1(b). In both

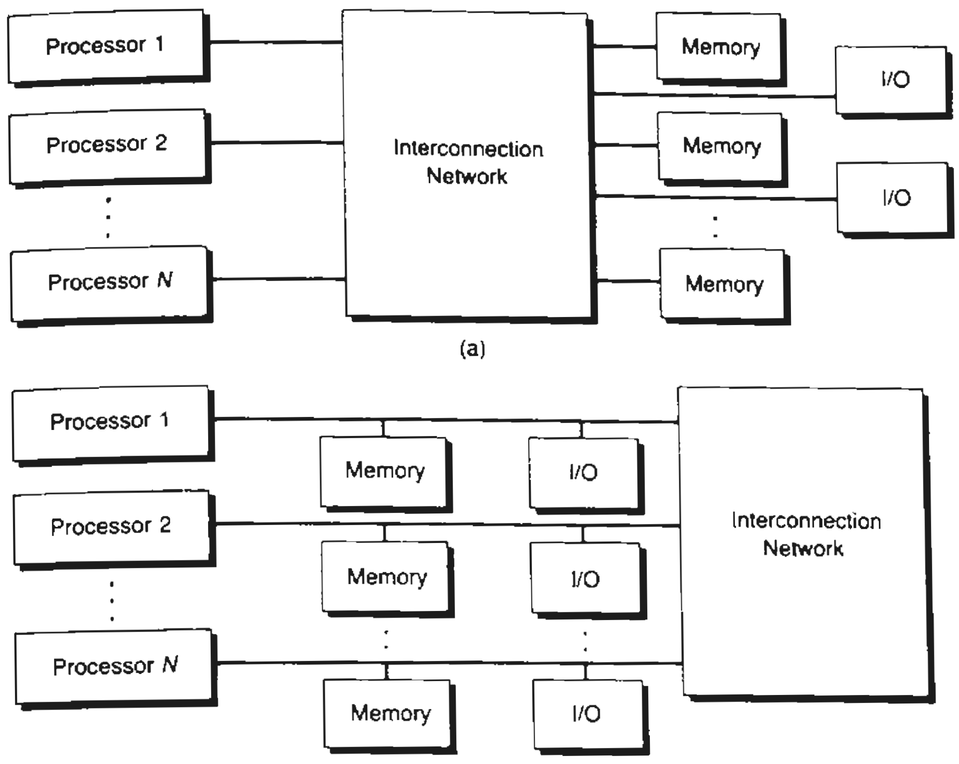

Fig. 6.1 Two multiprocessor structures:
(a) All memory and I/O are remote and shared; and
(b) All memory and I/O are local and private.

cases, because the system contains multiple processors, each capable of executing an independent program, the system fits Flynn's MIMD model.

In both systems depicted in Fig. 6.1 the processors cooperate by exchanging data through the interconnection system and by synchronizing activities. The shared memory in Fig. 6.1(a) provides a convenient means for information interchange and synchronization since any pair of processors can communicate through a shared location. The structure in Fig. 6.1(b) supports communication through point-to-point exchange of information. Obviously, multiprocessors can have any reasonable combination of shared global memory or private local memory. Figure 6.1 shows the extremes in the design space, and practical designs lie at the extremes or anywhere in between.

The main purpose of a high-speed multiprocessor is to complete a job faster by using several machines concurrently than can be done by using a single copy of the same machine. In some applications, the main purpose for using multiple processors is for reliability rather than high performance. The idea is that if any single processor fails, its workload can be performed by other processors in the system. Since the design principles of such systems are quite different from the principles that guide the design of high-performance systems, we do not address design for reliability in this text, but rather we limit our attention to issues related to performance.

When a multiprocessor is operating at peak performance, all processors are engaged in useful work. No processor is idle, and no processor is executing an instruction that would not be executed if the same algorithm were executing on a single processor. In this state of peak performance, all N processors of a multiprocessor are contributing to effective performance, and the processing rate is increased by a factor of N.

Peak performance is a very special state that is rarely achievable. There are several factors that introduce inefficiency. Among the factors are:

- The delays introduced by interprocessor communications;
- The overhead in synchronizing the work of one processor with another;
- Lost efficiency when one or more processors run out of tasks;
- Lost efficiency due to wasted effort by one or more processors; and
- The processing costs for controlling the system and scheduling operations.

Both scheduling and synchronization are sources of overhead on serial machines. In citing these factors together with the other factors, we are citing how they degrade multiprocessor performance beyond the effects that may already be present on individual processors.

A high-performance vector processor is free from many of the problems, but it does suffer from lost performance because it is unable to keep all of the processing units busy. This latter problem arises particularly when a computation is not easily implemented as a sequence of vector operations performed on highly structured, densely stored data.

The architect who designs and builds a multiprocessor must pay close attention to the sources of inefficiency exposed here. They can lead to serious degradation in performance. For example, if the combined inefficiencies produce an effective processing rate of only 10 percent of the peak rate, then ten processors are required in a multiprocessor system just to do the work of a single processor.

Fortunately, for a small number of processors, careful design can hold the inefficiency to a low figure, but inefficiencies tend to climb as the number of processors increase. There is a point where adding additional processors can lengthen, not shorten, computation time.

The fact that inefficiency tends to grow with the number of processors is the underlying reason why many commercial offerings of multiprocessors have a small number of processors, such as 4, 8, or 16. The fastest machines are built from the fastest devices available and have relatively few processors.

Consider, for example, the Cray XMP, a four-processor version of the Cray I. Another example is the IBM 309X family for which systems with up to six processors are available. Both of these implementations start with very high speed devices and use architectural techniques such as cache and pipelining to produce very high performance single processors for their respective markets.

Users of these machines may have workloads or individual problems whose needs exceed the capacity of a single machine. Additional performance is not readily available from faster versions of the same machine because the machines are already at the limits imposed by architecture and device technology. An effective way to attain small multiples of performance improvement is to group together two or four identical processors.

Some computer architects take note of a cost characteristic mentioned in Chapter 1. The discussion there indicates that high-speed device technology is much more expensive than lower-speed technology.

Moreover, with today's devices the cost of fast devices tends to grow faster than the performance benefit of the increased device speed. Hence, the cost per unit of computing power tends to be greater for high-end machines than for low-end machines, although this trend is technology dependent and could change over time. Nevertheless, when lower-speed technology has a cost advantage, we have an opportunity to create a cost-effective high-performance system by combining hundreds or thousands of slow-speed processors built with low-cost devices.

The cost advantage of using low-cost technology is balanced by the degradation in efficiency that inevitably occurs as the number of processors increases. If the degradation due to the large number of processors exceeds the cost advantage of the low-cost technology, then there is no particular advantage to using hundreds of slow processors over using a few very fast processors.

Moreover, the complexity of programming a machine with hundreds of processors far exceeds the complexity of programming a single processor or a computer system with just a few processors. Consequently, although economics might enhance the attractiveness of a machine with hundreds of low-speed computers, the advantage of this structure disappears if efficiency is not held high.

Thus, there is no particular magic in the parallelism of a multiprocessor. The parallelism yields a useful benefit when it successfully produces higher performance. When the parallelism cannot be tapped effectively, it simply adds to the system cost and complexity. In such a case, the end user is best served by reducing the parallelism to a point where the parallelism available can be used effectively. Whether there are 10, 1000, or 1,000,000 processors, it is bad practice to squander processing power. The argument that "processors are cheap" is irrelevant if, by using fewer processors, performance goes up.

In the next section we address the question of efficiency more carefully, especially considering the ratio of the time spent executing useful instructions compared to the time spent communicating with other processors.

6.2 Multiprocessor Performance

The point of this section is to analyze the performance benefit of multiple processors in the face of overhead incurred to create parallelism. The models studied are variations of models introduced by Indurkhya, Stone, and Xi-Cheng [1986]. This section shows that performance benefits strongly depend on the ratio R/C, where R is the length of a run-time quantum and C is the length of communications overhead produced by that quantum. The ratio expresses how much overhead is incurred per unit of computation. When the ratio is very low, it becomes unprofitable to use parallelism. When the ratio is very high, parallelism is potentially profitable. Note that a large ratio can be obtained by partitioning a computing job into relatively few large pieces, and that the amount of parallelism for such a ratio might be much smaller than the maximum available.

The ratio R/C is a measure of *task granularity*:

- In *coarse-grain* parallelism, R/C is relatively high, so each unit of computation produces a relatively small amount of communication; and
- In *fine-grain* parallelism, R/C is very low, so there is a relatively large amount of communication and other overhead per unit of computation.

Coarse-grain parallelism arises when individual tasks are large and overhead can be amortized over many computational cycles. Fine-grain parallelism usually provides opportunities to perform execution on many more processors than can fruitfully support coarse-grained parallelism. The idea of fine-grain parallelism is to partition a program into increasingly smaller tasks that can run in parallel. At the ultimate limit, each individual task may be as small as a single operation. More commonly, however, a fine-grained task contains a small number of instructions.

The programmer seeking maximum performance is strongly tempted to partition a problem into the finest possible granularity to create the maximum amount of parallelism. But if the maximum parallelism also has the maximum overhead, it is not clear that maximum parallelism leads to the fastest solution.

The main reason for the presentation of the performance models in this section is to show the pervasive role of the R/C ratio on performance. The discussion that follows shows how a fine-grain partition that happens to have a low R/C ratio produces poorer performance than a much coarser partition with a higher *R/C* ratio. Hence the much higher parallelism of the fine-grain partition need not produce higher net speed.

The purpose of presenting a number of different performance models to make this point is that no one model is truly representative of multiprocessors or of multiprocessor algorithms. We consider a number of different variations of the basic model to cover a variety of program behaviors and multiprocessor architectures. In every case, the role of *R/C* is the same. Small ratios lead to poor performance because of high overhead. Large ratios usually reflect poor exploitation of parallelism. For maximum performance, it is necessary to balance parallelism against overhead. The only difference from model to model is the point where the two factors become balanced.

Architects have long debated the relative qualities of fine and coarse granularity. For SIMD machines, the GF-11 is a coarse-grained machine whose individual processors can sustain a peak rate as high as 20 Mflops. The Connection Machine (CM-1) is an SIMD machine whose 1-bit processors are better suited to fine-grained tasks and whose performance stems from the massive number of processors rather than from the computational power of an individual processor.

What reasoning led the architects of one machine to seek such a vastly different solution than did the architects of the other machine? The range of applications is the primary motivation for the difference. The Connection Machine is designed to exploit parallelism of tasks such as image analysis, in which a significant portion of the work is characterized by fine-grained tasks. The GF-11, which is designed for much larger-grained tasks, would be burdened by overhead if the tasks carried the additional overhead attributable to fine granularity. Thus the architects of each machine attempted to match granularity to the applications for the machine.

At one end of the multiprocessor scale are the Cray multiprocessors, such as the Cray XMP—a four-processor system in which each processor is a Cray I supercomputer. Under ideal circumstances, communication in this system occurs only at the end of major phases, which might well be every few million or few billion instructions.

Smaller granularity is evident on microprocessor-based multiprocessors such as the Cosmic Cube and a number of commercial versions of this hypercube-based design. These machines typically use 64 to 256 copies of a high-performance 32-bit microprocessor. The different granularity biases the machines somewhat to different application programs.

The remainder of this section is devoted to performance models. In each model, observe how the ratio *R/C* determines the strategy that achieves the optimum performance. To simplify the models, we have generally ignored the effects of synchronization and contention except as crudely approximated by the models. In practical systems, the effects ignored here tend to lower performance from that predicted by these models. In most instances, the best way to compensate for the unmodeled effects is to increase the granularity of tasks.

6.2.1 The Basic Model—Two Processors with Unoverlapped Communications

For the first model, consider an application program that contains M tasks. Our objective is to execute this program at maximum speed on a system with N processors. For simplicity, we first consider a system with just two processors and then let the number of processors increase. To model performance we need to characterize the combination of execution time and overhead that will be incurred.

Let us make the following assumptions to obtain our initial results. Subsequently we relax the assumptions and see how the performance changes. Specifically, we assume that

1. Each task executes in R units of time; and
2. Each task communicates with every other task at an overhead cost of C units of time when the communicating tasks are not on the same processor, and at no cost when the communicating tasks are coresident.

We have various choices of how to execute such an application on a two-processor system. We can assign all tasks to one processor and ignore the second processor, which is a solution that minimizes communication overhead but fails to take advantage of available parallelism, or we can partition the tasks to the two processors in any combination. If the tasks are split across the processors, then the total execution time is a combination of the time spent in execution and the time spent engaged in overhead activities. Although we use the notation C as if C were exclusively due to communication, it is convenient to lump overhead from all sources into C.

To some extent, overhead can be overlapped with computation, especially if processors can perform communication through input/output ports while executing concurrently. However, not all sources of overhead can be hidden by overlapping with computation. Processors can contend for shared data or shared communication paths, and they may be idle during synchronization periods. Therefore, we assume that some portion of overhead operations lengthen total processing time because overhead cannot be fully overlapped with computation. In this case the equation that describes total processing time is the following:

$$\text{Execution Time} = R\,\text{Max}\,(M - k, k) + C(M - k)k \qquad (6.1)$$

Equation (6.1) expresses execution time as the sum of two terms, one attributed to run time and one to communication and other overhead. The run time for two processors is the larger of the run times experienced and is therefore the larger of $R(M - k)$ or Rk when k tasks are assigned to one processor and $M - k$ to the other. The second term models overhead to be proportional to the number of pair-wise communications that must take place as a function of how tasks are partitioned to the two processors. Note that the first term is a linear function of k, and the second term is a quadratic function of k.

What is the minimum execution time for Eq. (6.1) as a function of k? That is, how shall we assign tasks to two processors to produce the minimum execution time? Figure 6.2 shows a graphic way of finding a solution. The answer for this model is to assign all tasks to one processor if R/C is below $M/2$, or split

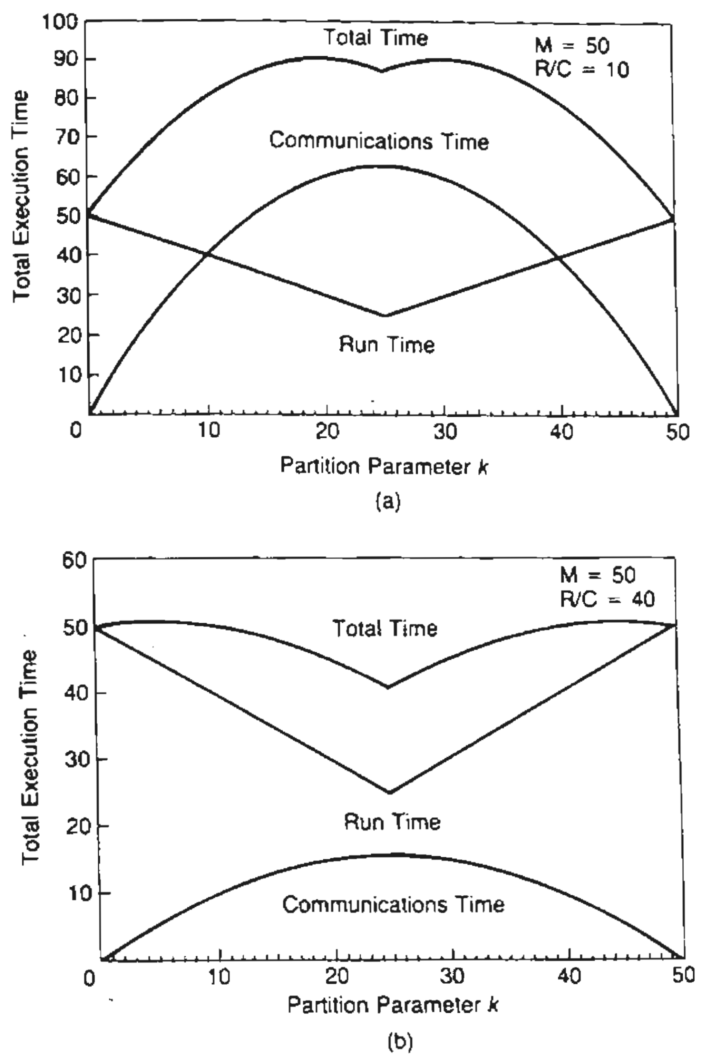

Fig. 6.2 Parallel execution time for two different R/C ratios:
(a) Optimum partition parameter $k = M/2$; and
(b) Optimum partition parameter $k = 0$.

the tasks evenly between two processors if R/C exceeds that threshold. That is, either $k = 0$ or $k = M/2$. (If k is odd, then make k as close to $M/2$ as possible.)

Figure 6.2 shows the two different cases that arise for the different values of the R/C ratio. The first term of Eq. (6.1) is piece-wise linear, and Fig. 6.2(b) shows that this term looks like the letter V because it is symmetric about the point $k = M/2$. In this figure, when the piece-wise linear term is added to the quadratic term, the resulting figure has a minimum at $M/2$.

In Fig. 6.2(a), the minimum occurs at $k = 0$. The minimum has to be at an extreme point in the region $0 \le k \le M/2$ because the quadratic curve $k(M - k)$ is concave downward, and, after adding a linear term to this curve, the concavity is unchanged. A curve that is concave downward has its minimum at one of its endpoints. The endpoint of the curve at $k = 0$ (or at $k = M$) is the minimum when $R/C < M/2$; otherwise the minimum occurs at $k = M/2$.

6.2.2 Extension to *N* Processors

Now let's consider what happens when there are N processors. In this case, we assign k_i tasks to the ith processor. The generalization of Eq. (6.1) becomes

$$\begin{aligned} \text{Execution Time} &= R \text{ Max } (k_i) + \frac{C}{2} \sum_i k_i(M - k_i) \\ &= R \text{ Max } (k_i) + \left(\frac{C}{2}\right)\left(M^2 - \sum_i k_i^2\right) \end{aligned} \qquad (6.2)$$

The first term counts the longest running time among the N execution times. To that time is added the overhead from the second term. That term counts the number of distinct pair-wise links between k_i tasks and $M - k_i$ tasks, each of which contributes an amount C to the total time. The second term in Eq. (6.2) is quadratic just as in Eq. (6.1).

If the reasoning used to analyze Eq. (6.1) holds for this equation, then we expect that the minimum value is for an extreme assignment, and indeed this is the case. Either all tasks are assigned to a single processor, or they are distributed "evenly" across all processors. By "evenly," we mean that if M is a multiple of N, then each processor receives M/N tasks. Otherwise, all but one processor receives the integer ceiling of M/N tasks, and one processor receives whatever is left over. This assignment does not necessarily use all N processors. For example, when there are 19 tasks and six processors, the assignment places 4 tasks on four processors and 3 tasks on a fifth processor, leaving no tasks assigned to the sixth processor.

To show that the even distribution produces a local minimum, assume that k_1 has the maximum number of tasks assigned to it, and show that an assignment in which two processors receive fewer than k_1 tasks can be changed to an assignment with a lower cost, as computed by Eq. (6.2).

For example, assume that both k_2 and k_3 satisfy $k_1 > k_2 \geq k_3 \geq 1$. Consider the assignment that shifts one task from the third processor to the second processor and examine how the cost changes as per Eq. (6.2). The first term does not change because the change does not affect the maximum number of tasks assigned to a processor. The value of the second term is reduced, however, by the amount $C(k_2 - k_3 + 1)$. This assignment produces higher performance, and we can iterate this improvement process until no more than one processor has less than the maximum number of tasks assigned to it.

Equation (6.2) has a threshold for an assignment, just as Eq. (6.1) has, and by a remarkable coincidence the thresholds are identical! We must compare the even assignment of tasks to the assignment that places all tasks on one processor. The latter assignment is preferred when R/C is sufficiently small.

The difference in costs of the "even" distribution to N processors and a 1-processor assignment is given by

$$\text{Time Difference} = \frac{RM}{N} + \frac{CM^2}{2} - \frac{CM^2}{2N} - RM \tag{6.3}$$

where the first three terms form the cost of the even distribution of tasks and the last term is the cost of assigning all tasks to one processor.

To simplify the analysis, we have ignored values of M that are not exact multiples of N. To solve for the threshold value of R/C, we set the value of Eq. (6.3) to 0. By removing a factor of M and then grouping terms by coefficients R and C, we can remove another factor of $(1 - 1/N)$. This yields the equation

$$\text{Time Difference} = \frac{CM}{2} - R = 0 \tag{6.4}$$

or

$$\frac{R}{C} = \frac{M}{2} \tag{6.5}$$

This model shows that if R/C is greater than the threshold $M/2$, then an even distribution of tasks to as many processors as are available will produce the best time. On the other hand, if R/C is below that threshold, then no matter how many processors are available, no assignment produces a faster time than the assignment that uses only one processor. Here is a situation in which the role of overhead becomes quite clear.

Unless overhead is kept below a certain percentage of execution time, parallel execution cannot be beneficial. If this model holds for a parallel algorithm and architecture, then the control of overhead costs is absolutely essential for parallelism to be successful.

Although this analysis has looked at performance rather than costs, R/C determines the point at which parallelism is cost-effective. Even when R/C is sufficiently high to warrant parallelism, the performance gain is diminished by the second term of Eq. (6.2). The speedup attributable to parallelism is the ratio of the time to run on one processor to the time expressed by Eq. (6.2). This is approximately

$$\begin{aligned}\text{Speedup} &= \frac{RM}{\left(\frac{RM}{N} + \frac{CM^2}{2} - \frac{CM^2}{2N}\right)} \\ &= \frac{R}{\left(\frac{R}{N} + \frac{CM(1 - 1/N)}{2}\right)} \qquad (6.6) \\ &= \frac{\frac{RN}{C}}{\left(\frac{R}{C} + \frac{M(N-1)}{2}\right)}\end{aligned}$$

If the first term of the denominator is large compared with the second, then the speedup is proportional to N. This requires M and N to be small and for R/C to be large. If parallelism is increased to the extent that the denominator is dominated by its second term because N is very large, the speedup is proportional to R/CM, which does not depend on the number of processors. Hence, as N increases, the speedup approaches a constant asymptote.

At this point each processor added to the system brings extra cost while yielding negligible performance benefit. Even though performance can improve incrementally as processors are added, the diminishing returns in performance are not worth the added cost. The number of processors should not be increased beyond some maximum that is a function of cost and the ratio R/C.

This model is a general picture of how granularity and overhead affect the performance gain of a multiprocessor, and it gives some indication of the importance of minimizing overhead and selecting the right granularity. It is only one model, however, and it cannot encompass the full spectrum of actual applications.

Let us alter the model in various ways and observe how the findings change. In general, we discover that R/C plays a critical role, regardless of the model. In some cases, there is the same type of threshold in which the best solutions are extreme. That is, use all available processors or just one processor, depending on the value of R/C. In some models, the extreme solutions are not the best. The best solutions for these models distribute work among several processors, but do not use all processors because the use of too many leads to performance degradation and extra cost. Moreover, in the general case, work need not be distributed evenly to achieve the optimum performance.

6.2.3 A Stochastic Model

Consider what happens when all tasks are not equal in execution time. The leading term in Eq. (6.2) is smallest when all processors run for equal lengths of time, so the objective is to scatter tasks among processors so that all processors are occupied for equal times. If this is not possible, the maximum running time among the processors should be as short as possible.

The second term in Eq. (6.2) is smallest when tasks are distributed as unevenly as possible. Consequently, among all ways of distributing tasks to processors so that processors have nearly equal running times, find a distribution in which the number of tasks assigned to each processor is as uneven as possible. That is, find schemes that assign as few or as many tasks per processor as possible, subject to the requirement that the total workload on a processor be equal to a given amount.

In this model, the best assignment need not be the most evenly distributed workload. If the workload is slightly uneven, it may become possible to assign tasks to processors in such a way that overhead is greatly diminished. That is, a small increase in the linear first term of Eq.(6.2) can be more than balanced by a large decrease in the quadratic second term.

A stochastic variation of the deterministic model presented here appears in Indurkhya, Stone, and Xi-Cheng [1986]. Instead of having all execution and communication times as fixed constants, the model assumes that the times are independent and identically distributed random variables with a mean R for the running times and a mean C for the communication times. To solve the model, Indurkhya *et al.* appeal to the central limit theorem and the additional assumption that

$$E\left[\text{Max}\left\{\sum_{i=1}^{k} r_i, \sum_{i=k+1}^{M} r_i\right\}\right] = \text{Max}\left\{E\left[\sum_{i=1}^{k} r_i\right], E\left[\sum_{i=k+1}^{M} r_i\right]\right\} \qquad (6.7)$$

The E in Eq. (6.7) denotes the expected value. Equation (6.7) says that the maximum of a set of expected values of sums of independent and identically distributed random variables r_i, the running times of the tasks, is equal to the expected value of the maximum of the sums. With these two assumptions, the model reduces to the deterministic model expressed by Eq. (6.2), and the results are identical.

The assumption underlying Eq. (6.7) is actually false, as is stated by Indurkhya *et al.*, but the point is that when the equation breaks down, it is close enough to being correct that the results produced are reasonably accurate. If one of the summations in Eq. (6.7) has many more summands than any other, then almost surely it has the maximum expected value, and its expected value is the value of both sides of Eq. (6.7). If the two summations have almost the same number of terms, then it is possible for the left-hand side of Eq. (6.7) to select one summation and the right-hand of Eq. (6.7) to select the other summation, but the values of the summations will be fairly close, so that Eq. (6.7) is approximately if not exactly correct. Equation (6.2) covers the assignment to $M = 2$ processors. It generalizes to $M > 2$.

Nicol [1989] explored the model more deeply and discovered that the results reported by Indurkhya *et al.* can be proved to be true in some instances without relying on Eq. (6.7). Indeed, the model appears to be robust in the sense that small perturbations in the underlying assumptions do not alter the gross conclusions from the model, although specific details in the conclusions may change.

6.2.4 A Model with Linear Communication Costs

Let us examine a model that is less drastic with regard to communication costs to show a more optimistic result with regard to parallelism. Our first model assumes that each task communicates with every other task, and, as a consequence, the communications overhead grows quadratically as the number of processors increases. This is the case when each task sends unique information to every other task, but such a program structure is very poorly suited for multiple processors. Some programs may well have this structure, and if so, our results suggest how much speedup one can expect and at what cost. But there are surely many other programs better suited for parallel computation on multiprocessors. We need to know the performance potential for such programs and how to achieve it. What is rather surprising is that the analysis is remarkably similar and uncovers a rather similar optimal strategy, although the speedup available is greater.

For this model, assume that the cost of communication is proportional to the number of processors, not to the number of tasks assigned remotely. This model holds if a task has to communicate with all other tasks but sends the same information to all other tasks. Then the information has to be sent only once to each processor, and after it reaches a remote processor it can be sent from task to task within that processor for no charge.

In this model the cost of an assignment on N processors becomes

$$\text{Execution time} = R \text{ Max } (k_i) + CN \tag{6.8}$$

For each value of N, the first term depends on the assignment but the second does not. This model produces the best time by distributing tasks evenly across all processors to make the first term approximately equal to RM/N. However, as the value N increases, the increase in the second term eventually becomes larger than the decrease in the first term, so there is a maximum value of N for which performance increases, and this is a function of R/C.

Since the best assignment produces a first term of approximately RM/N, the decrease in time in going from N to $N + 1$ processors is approximately

$$\begin{aligned}\text{Execution time decrease} &= RM\left(\frac{1}{N} - \frac{1}{N+1}\right) - C \\ &= \frac{RM}{N(N+1)} - C\end{aligned} \tag{6.9}$$

This decrease is negative, that is, it becomes a time increase when

$$\frac{R}{C} = \frac{N(N+1)}{M}$$

or equivalently when

$$N = \sqrt{\frac{RM}{C}} \tag{6.10}$$

The square root function in Eq. (6.10) is a disaster. We expect that M tasks can be done quickly on M independent processors, but this model says that because of communication costs, the effective parallelism is reduced to the square root of what we anticipated. The bad news is mitigated somewhat by a high R/C factor, so coarse granularity is desirable here, but its effect is also diminished by a square root factor.

The news is even more pessimistic if we consider the cost of the extra processors in relation to their benefit. Given that the time no longer decreases when we reach the threshold given in Eq. (6.10), long before N becomes that large, we have reached the point at which the cost of adding an extra processor is not justified by the benefit gained. Thus a problem with 10,000 tasks that fits this model may well run faster with up to 100 processors and might be economical with at most 10 processors.

This model differs from our original model in the second term. In the original model, Eq. (6.2), the cost of the second term grows quadratically with the constant M. Contributions to time vary inversely with N. For large N, execution time approaches $CM^2/2$, which does not increase as N increases. Because both the first and second terms grow smaller with N in the original model, execution time decreases for all N.

In the present model, the second term grows linearly with N, and this accounts for the threshold for N above which performance degrades. The two models tell us that the penalty for overhead exists, and it manifests itself by limiting the effective use of parallelism in some way.

6.2.5 An Optimistic Model—Fully Overlapped Communication

Perhaps the models described thus far are too pessimistic. After all, they all incur an overhead penalty for communication since none provides a means for overlapping overhead with useful and necessary computation. We have argued that in practical systems some overhead cannot be masked because contention, finite communications bandwidth, and synchronization each make their own contributions to elapsed computation time, although in the best circumstances some overhead penalties can be successfully overlapped with useful computation to reduce the overhead penalty.

Let us develop an optimistic model in which overhead potentially can go to zero if overlapped with computation. We simply alter our model in Eq. (6.2) to permit the overhead in the second term to be overlapped as much as possible with the first term. The equation becomes

$$\text{Execution Time} = \text{Max}\left\{\text{Max}\,(k_i),\ \frac{C}{2}\sum_i k_i(M - k_i)\right\} \qquad (6.11)$$

For two processors, the situation described by Eq. (6.11) is depicted in Fig. 6.2. The piece-wise linear line expresses the contribution of the first term, and the quadratic curve expresses the contribution of the second term. Their intersection is the minimum value of the maximum function expressed in Eq. (6.11). At this point the execution time is just long enough to mask completely the overhead that is occurring concurrently.

This model is obviously optimistic because it is rather unlikely that overhead can be fully overlapped with processing. Nevertheless, we can compute where the threshold occurs. For two processors, we seek the point of intersection of the linear and quadratic curves in Fig. 6.2. This occurs at the point

$$R(M - k) = C(M - k)k \qquad (6.12)$$

which occurs at

$$k = \frac{R}{C} \qquad (6.13)$$

with k restricted to the range $1 \le k \le M/2$. If we substitute Eq. (6.13) into Eq. (6.11) the computation time becomes $R(M - R/C)$, and the speedup is $1/[(1 - R/CM)]$. Since k is restricted in range for Eq. (6.13), the equivalent restriction on R/C is that $1 \le R/C \le M/2$. For R/C in this range, the speedup for two processors lies between 1 and 2 and is maximized when $R/C = M/2$, the same value obtained in the first model.

At the maximum speedup, the tasks are evenly divided among the processors, that is, $k = M/2$. As R/C decreases toward 1, the speedup falls off toward unity, and the optimum task distribution becomes more skewed. Hence, this model also depends on R/C, but it is more optimistic in its performance predictions because all or a substantial portion of overhead can be overlapped with computation if R/C is high enough.

For N processors, the overlapped-overhead model is easy to analyze because of the results reported here. For any given maximum value of k_i that determines the contribution of execution time, the even distribution of tasks to processors as defined earlier produces the minimum communication time. Hence, the best possible execution time for fully overlapped communication occurs when

$$\frac{RM}{N} = \frac{CM^2}{2}\left(1 - \frac{1}{N}\right) \tag{6.14}$$

which for large N occurs roughly when

$$\frac{R}{C} = \frac{NM}{2} \tag{6.15}$$

In this case, for a minimum total time, the number of processors as a function of R/C and M is given by the function

$$N = \frac{2R}{CM} \tag{6.16}$$

and the optimum choice for the number of processors is inversely proportional to the number of tasks available.

As the available parallelism grows, the best policy is to use increasingly fewer processors. For small N, we cannot neglect the $1/N$ term in Eq. (6.14), and we obtain slightly different but consistent results. For $N = 2$, Eq. (6.14) produces a minimum-time solution when $M/2 = R/C$, which is consistent with our previous findings.

The fact that the number of processors decreases with the available parallelism in this model is clearly the result of overhead time climbing M times faster than execution time. The effect of overlapping overhead with computation time is actually more pessimistic than we imagined because this model makes elapsed time totally dependent on communication overhead time when run time is smaller than communication time. Hence, it is absolutely essential to keep communication time no greater than execution time if there is to be speedup.

6.2.6 A Model with Multiple Communication Links

A common assumption in all previous models is that parallelism allows run time to be overlapped in several processors, but overhead operations accounted by the term with coefficient C are done sequentially. If the overhead operations are strictly limited to communications costs, then this model holds for systems in which there is a single communications channel common to all processes. This is the case when all processors are connected to a single bus or ring or when all processors access the same shared-memory cell in an exclusive-access manner.

It is perfectly possible to replicate communications links and other architectural features that contribute to the overhead bottleneck of the second term. In so doing, the factor C is not a constant, but itself becomes a function of N. For example, consider a model in which every process has to communicate with every other process. Our original estimate for run time is Eq. (6.2).

If we allow communication links to increase with N so that each processor has a dedicated link to every other processor, then communication operations can be overlapped among themselves. However, even with $O(N^2)$ links installed, we still cannot support more than $O(N)$ concurrent conversations because each processor can talk or listen only to one other processor at a time.

In this case, we can divide the second term of Eq. (6.2) by N, and we obtain

$$\text{Execution Time} = R \operatorname{Max}(k_i) + \frac{C}{2N}\sum_i k_i(M - k_i) \tag{6.17}$$

Equation (6.17) assumes that a processor is either computing, communicating, or idle, and that the total cost of communications decreases inversely with N because up to N conversations can be held concurrently. The idle time in part is due to the fact that early finishers have to wait for late finishers.

Both terms of Eq. (6.17) tend to decrease inversely with N. The form of Eq. (6.17) is very similar to Eq. (6.2) except for a factor of N in the second term. An even distribution minimizes the first term but not the second term. It follows that Eq. (6.17), like Eq. (6.2), is minimized by assigning tasks as evenly as possible, so that all except possibly one processor are given the maximum number of tasks. Under such an assignment, the execution time for Eq. (6.17) becomes

$$\text{Execution Time} = \frac{RM}{N} + \frac{CM^2}{2N}\left(1 - \frac{1}{N}\right) \tag{6.18}$$

Parallelism is useful in this case until execution time fails to decrease as new processors are added. This occurs when the following equation is negative.

$$\text{Execution Time Decrease} = \frac{RM + \dfrac{CM^2}{2}}{N(N+1)} - \frac{\left(\dfrac{CM^2}{2}\right)(2N+1)}{[N(N+1)]^2} \tag{6.19}$$

By removing a factor of $[M/N(N + 1)]$ and letting N become very large, Eq. (6.19) reduces to

$$\text{Execution Time Decrease} = \left[R + \left(\frac{CM}{2}\right)\left(1 - \frac{2}{N}\right)\right]\left(\frac{M}{N(N+1)}\right) \tag{6.20}$$

which is positive for $N > 2$, and so execution time improves for all N, except possibly for small N.

To discover if N processors yield a better time than does one processor, compare Eq. (6.18) with RM, the time for one processor. These times are equal when

$$RM = \frac{RM}{N} + \left(\frac{CM^2}{2N}\right)\left(1 - \frac{1}{N}\right) \tag{6.21}$$

The breakeven point occurs when

$$\frac{R}{C} = \frac{M}{2N} \tag{6.22}$$

In this case the granularity factor R/C and N are inversely related at the breakeven point. Hence, the larger that N is, the smaller the granularity that we can permit at the breakeven point. At breakeven, however, the parallel machine is a gross failure in terms of cost/performance. Its total performance for N processors is identical to that of a single processor, yet its cost is higher by a factor of $O(N)$ for processors and $O(N^2)$ for communication links. We never want to operate a parallel system at breakeven!

The point of this example is that by increasing the bandwidth of the communication links, we can permit smaller granularity than is otherwise possible. However, the smaller granularity comes at an expense that rises faster than the increase in processing cost. Whether or not the speed obtained by the higher bandwidth communications is worth the cost depends very strongly on the technology available for processor-to-processor communications.

To summarize the findings of the models presented in this section, we have discovered:

1. Multiprocessor architecture produces an overhead cost that is an additional burden not present in serial processors and vector (or other single instruction-stream) architectures. The overhead cost includes the cost of scheduling, contention for shared resources, synchronization, and processor-to-processor communications.
2. Although running time for a computational portion of a program tends to diminish as the number of processors working on that program increases, the overhead costs tend to grow with the number of processors. In fact, it is possible for overhead costs to grow faster than linearly in the number of processors.
3. The ratio R/C is a measure of the amount of program execution (running time) per unit overhead (communication time), within a program implementation on a specific architecture. The larger this ratio, the more efficient the computation because a relatively smaller proportion of time is devoted

to overhead as this ratio increases. However, if the ratio is made large by partitioning a computation into a few large pieces instead of many small pieces, the parallelism available is greatly reduced, which limits the speedup that can be attained on a multiprocessor.

We clearly have a dilemma. On the one hand, R/C has to be small to create a large number of potentially concurrent tasks, and on the other hand, R/C has to be large to prevent the overhead costs from becoming excessive. Because of the dilemma, we cannot expect to build fast multiprocessors simply by expanding the number of processors as much as technology allows.

There is some maximum number of processors that is cost-effective, and that number depends a great deal on the architecture of the machine, on the underlying technology (especially communications technology), and on the characteristics of each specific application.

6.2.7 Multiprocessor Models

The multiprocessor challenges the computer architect and the algorithm designer somewhat differently. The computer architect must produce a system for which R/C is acceptably high and provide a number of processors that can be used effectively at that ratio. The algorithm designer has a different problem.

Given a fixed system with N processors and a ratio R/C that reflects an achievable ratio of running time per unit overhead, how can an application be partitioned and executed on the multiprocessor architecture to make the most effective use of resources? The algorithm designer has to partition the application across the multiprocessor and must choose a granularity that balances useful parallel computation against communications and other overhead.

For some applications the most effective solution might not use all of the processors available. Fewer processors might complete the job earlier or at lower cost. In essence, we are trying to determine if it is better to plow a field with one ox, four horses, or 1024 chickens. The solution with the maximum parallelism is not always the fastest.

Most people take as an act of faith that one might as well use as many processors as available if there is work to be done. However, some models discussed in this section show that computation speed can eventually decline as processors are added. So maximum parallelism is not synonymous with maximum speed. Moreover, the multiprocessor is somewhat less effective at producing speed at reasonable cost than are several techniques described earlier in the text.

For example, cache memory boosts the effective speed of all of central memory, yet only a relatively small fraction of memory actually needs to run at cache memory speed. Hence, there is a performance leverage in using a cache. You pay for a small fraction of what you obtain.

Similarly, pipeline computers improve performance in proportion to the number of stages in the pipeline. In the best case, an *N*-stage pipeline achieves an *N*-fold speedup. But the *N*-fold speedup does not require an *N*-fold replication of hardware. Again, there is leverage in this type of architecture because by less than an *N*-fold increase of hardware, one obtains up to an *N*-fold improvement in speed.

In both cases the leverage is available because the item replicated is a bottleneck that leaves other system resources idle. By breaking the bottleneck the idle resources become available, and the total gain appears to be greater than the gain that can be attributed to the fixed bottleneck by itself.

For cache, the bottleneck is memory, specifically the frequently referenced areas of memory. For pipelines, it is some computational stage or critical register. Cache replicates memory; pipelines replicate storage cells and arithmetic units. But multiprocessors do not obviously offer the same leverage as do caches and pipelines. The component replicated is the full processor, not some critical portion of the processor. Moreover, we are likely to obtain less than proportionate return as we add processors.

Therefore, the design of multiprocessor architecture is far more challenging than the techniques we describe earlier. One cannot simply lash together 1000 processors and expect to obtain 1000-times improvement. In fact, performance improvements of only 100 to 200 might be all that could be achieved under favorable circumstances, and under less favorable circumstances, improvement might be only around 10 or less.

On the other hand, with a greater understanding of overhead costs, algorithms, and design approaches available, it is possible to construct efficient multiprocessors. Our analyses in this section strongly suggest that efficiency becomes limited as the number of processors increases. Perhaps an architecture with 4 to 16 processors can be viewed as "general purpose," but with 1K or 64K processors, almost surely the architecture is limited to applications for which the inherent parallelism is large and the granularity is in the range for which the architecture runs well.

Hoshino [1989] has performed a granularity study of programs that are operational on the PAX machine. The results of his study are consistent with the predictions of this model. He measured the actual computation time and communication overhead for various applications from timings taken on a 128-processor machine. These timings were then scaled for various numbers of processors, and various amounts of local memory. Because synchronization and communication tend to be unoverlapped in the PAX architecture, the basic model introduced early in this section tends to capture the performance of many of the PAX applications. Hoshino's general conclusion is that the speedup attainable on a 1000-processor machine is quite reasonable, provided that the granularity ratio *R/C* is high enough to make the overhead a negligible portion of the computation. His data indicate for each application how much local memory is required to attain a satisfactory granularity. To maintain high *R/C* for 1000 processors, it is necessary to scale the size of the problem upwards. Hence a 1000-processor machine can be as efficient as a 100-processor machine, if the problem solved by 1000 processors is sufficiently larger than the problem solved by 100 processors.

Efficiency is clearly a major concern in the design of multiprocessors. A design that uses 2*N* processors inefficiently cannot compete on a cost basis with a design that uses *N* identical processors twice as efficiently. The next section treats some of the more promising candidate architectures for multiprocessors.

6.3 Multiprocessor Interconnections

This chapter investigates the following leading candidates for multiprocessor systems:

- Bus-oriented systems;
- Ring networks;
- Crossbar-connected systems;
- Two- and three-dimensional meshes;
- Multilevel switched-network systems; and
- Hypercubes.

This is not an exhaustive, but rather a representative list of the possibilities. As we examine low-cost, low-bandwidth communications through high-cost, high-bandwidth communications, the system issues are fairly constant across the spectrum.

Our major conclusion is that the multiprocessor interconnection structure is felt most strongly by imposing a saturation point for system communications. Consequently, peak throughput is limited by the interconnection structure. For performance below saturation, the interconnection structure affects performance through the ratio of *R/C*. A good design is one that runs below saturation for typical workloads, and at a typical operating point, it produces high throughput by attaining a large *R/C* ratio. If for a particular workload, the interconnection network of such a design can be modified in some major way without altering throughput, then there is some flexibility in the set of interconnections that can be used for that workload. The architect seeks the least costly set of interconnections that achieves good performance over a large class of applications.

6.3.1 Bus Interconnections

Our discussion of performance stresses the need for efficiency and shows the important role of the ratio *R/C*. The simplest way to construct a multiprocessor that meets the efficiency goals is to connect the processors on a shared bus, which thereby provides shared global memory to all processors. Figure 6.3 illustrates the block diagram of such a system.

Each processor has access to a common bus. To this bus is attached the central memory, which is a global resource for all processors. Each processor, in addition, has a local memory and a cache memory. The local memory and local cache enable the processors to reduce their use of the shared bus and thereby limit the effects of contention on performance when processors have to go to shared memory.

If neither cache nor local memory were present, the cost of memory access would be relatively high, and, moreover, since all processes access memory frequently under these conditions, there could be severe contention at the bus, causing arbitration delays that reduce performance. So the long delays due to remote access coupled with additional delays due to contention effectively increase the value of C in the R/C ratio and thereby reduce speedup and the number of processors for which the scheme is effective.

The objective in using cache and local memory is to shorten the effective memory cycle and reduce the use of the bus so that one processor does not slow down another through bus interference. If together the local memory and cache reduce accesses on the bus by 90 percent (which should be readily achievable), then 10 times as many processors can share a bus at a given level of contention than in the system that has no local memory or cache. If the global accesses are reduced by 95 percent, the factor climbs to 20 times as many processors.

Historically, commercial releases of bus-based multiprocessors supported as many as 32 microprocessors. Above 32 processors, bus contention leads to degraded performance. Unfortunately, the present trends in technology tend to

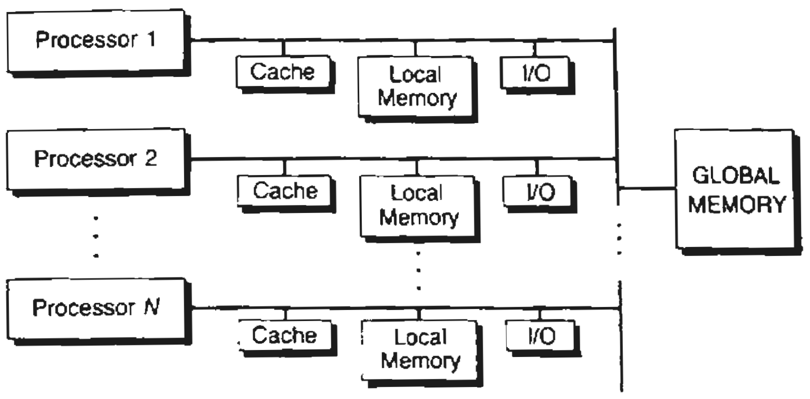

Fig. 6.3 A bus-connected multiprocessor.

reduce the number of processors that can be attached to a bus rather than increase it. The problem is that the mature interconnection technology available for such systems uses metal conductors, and the maximum clock speed of such buses is somewhere between 200 MHz and 300 MHz. The limiting factor turns out to be stray capacitance at the receivers on the bus, which causes reflections to travel backward toward the transmitters, from where they are again reflected toward the receivers where they are received as false pulses. The cure for this disease is to limit the rise-time of the transmitted signals, and therein lies the bandwidth limitation of the interconnection technology. Another possible cure is to reduce the physical dimensions of all devices and components in order to reduce stray capacitance. Although this is effective and is used successfully in the Cray III computer to support a 1 GHz clock cycle, it is also very costly.

If we restrict attention to low-cost high-volume technology, then the present trend is for individual components and processors to become faster every year while the clock cycle on the bus is fixed at an upper limit because of fundamental limitations. When processor clock rates were in the 5 MHz region, a 200 MHz limitation on bus clock rate did not overly constrain multiprocessor structure. With processor clocks reaching 50 MHz, 100 MHz, and 150 MHz in recent releases, it becomes clear that just a few active processors can saturate a bus. If the trend continues and bus technology remains based on metal interconnections, we are likely to see no increase in the number of processors in bus-based systems, and may even see a reduction as processors become faster relative to the bus.

Optical technology provides an alternative implementation of processor-to-processor interconnections. The technology is still developing, and it may be quite reasonable to use this technology for the backbone of a highly parallel bus-based computer system. Consequently, the future exploitation of bus-based architecture is intimately tied in the future success of optical buses.

Apart from the physical realization of a bus-based architecture, there are special issues involved in using caches in this architecture that we examine later in this chapter. The problems stem from the need to maintain consistency of data in all of the caches. If a shared item is changed in one cache and read by another processor, the second processor must be able to locate the new value of the shared variable. This forces the cache controllers to follow a protocol that guarantees that all loads and stores access the correct value of an item, regardless of whether that item is in local cache, remote cache, or global shared memory. The bus-based multiprocessor is a natural structure for building an effective cache-coherence protocol.

Usually such a protocol produces additional operations on the shared bus whose purpose is to guarantee cache consistency. If caches were not present, these operations might not be necessary. Hence, a cache architecture reduces bus accesses when the cache hit ratio is high, but the reduction is partially offset by additional bus transactions caused by the consistency protocol. With cache sizes large enough to reduce miss ratios to 1 percent, the potential impact on bus traffic is to reduce it 100-fold, thereby providing for as much as 100 times as many processors on a bus than could be supported without a cache. This calculation is overly optimistic because of the extra traffic to maintain cache coherence. Most of the traffic is required to communicate WRITEs so that all processors see updated data in case they need to have the most recent values of shared data. WRITEs account for 15 to 25 percent of memory operations, but in a cache-based processor that uses a store-in cache policy rather than a write-through cache policy, the percentage of memory operations that have to be communicated on a bus to other processors may drop to 5 to 10 percent. So cache may provide only a 10 to 20 times reduction in bus traffic rather than a 100-fold reduction, but the improvement from using a cache is a definite advantage in any case.

Technology plays a major role in making a bus-oriented multiprocessor practical, and, in fact, the bus presents an excellent opportunity for technology leverage. An *N*-processor system requires a bus whose bandwidth is on the order of *N* times that of a uniprocessor bus. Therefore, the bus bandwidth constrains the number of processors that can be interconnected as *N* increases.

If exotic technology is used only for the bus and its interfaces, but ordinary technology is used in the processors, then the cost of the exotic technology can be held fairly low, while the gain due to its use is amplified by greatly increasing the number of processors on the bus. Consequently, it may be feasible to use bus interconnections that run perhaps 100 times faster than basic processor technology and are capable of supporting 1000 processors. As we suggested earlier, a possibility for the future is to use optical links and gallium-arsenide transmitters and receivers whose information rate is in the 1 GHz to 10 GHz region.

But exotic technology can also work against the architect. If it can be used in the communication link, then equivalent technology might well be used throughout the system, boosting basic throughput in each processor by perhaps a hundredfold. In this case, perhaps only 10 super-technology processors can do the work of 1000 low-technology processors with a super-technology bus. The 10-processor, all-super-technology system might well be more cost-effective than the 1000-processor system because it is likely to be more efficient and less complex. The computer architect has to evaluate where and how to use exotic technology, carefully considering reasonable alternatives rather than committing arbitrarily to a specific use of the technology in a particular architecture.

Note that the bus is only one potential bottleneck in the bus-oriented multiprocessor. The shared memory is another one. As bus bandwidth increases, performance is eventually limited by the bandwidth of the shared memory. Because processors synchronize their activities by reading and writing shared memory cells, as the number of processors increases, there is a tendency for some shared cells to receive an increasing proportion of the memory references.

For example, consider a single memory cell that controls the execution of N processors by acting as a barrier. Processors wait at the barrier until all processors have reached it. Then they are free to continue. The barrier cell can be initialized to the value N, and, as each processor reaches it, the cell is decremented. When the cell is decremented to 0, all processors are released.

If the shared cell is accessed by one processor at a time, then clearly the time required for the barrier to go from N to 0 is $O(N)$ time. If the processors executing in parallel are performing some function that requires constant time, then for sufficiently large N the barrier itself becomes a bottleneck of the computation and greatly limits the useful work performed by the system.

To overcome the bottleneck in the shared memory, it is necessary to seek creative solutions in technology, architecture, or algorithms:

- *Technology:* use very high-speed devices for shared memory or move to an exotic memory technology that supports multiple simultaneous accesses.
- *Architecture:* design a system with high-bandwidth architectural support for sharing and control.
- *Algorithms:* for specific applications, seek means to distribute control to reduce or eliminate bottlenecks at centralized control variables.

All of the approaches are potentially viable. Any one approach may be sufficient to create a system of the desired performance. Relatively few ideas have been implemented and evaluated, and many opportunities for advances still exist.

Returning to bus-based interconnections, consider what techniques are available for bus implementation. The highest-speed electrical buses must be very short. This limitation is strictly a matter of physics because high speed implies fast changes of voltage and current. Such physical quantities are limited in their switching speed by capacitance and inductance. To hold these quantities small requires small physical distances because capacitance and inductance are proportional to conductor length.

Signal fidelity also diminishes when signals are sent over long distances, and the degradation in fidelity increases the probability of error during transmission. Therefore, if a bus is long or has other characteristics that slow transmission or degrade signal quality, the bandwidth of such a bus is lower than that of a short bus with excellent signal qualities. Yet another problem is crosstalk noise stemming from mutual interference from adjacent signals. This too grows with physical distance.

The problem is that as the number of processors tied to a bus increases, most electrical buses suffer degradation that tends to reduce bandwidth. Hence, not only does each processor have to share the bus bandwidth with $N - 1$ other processors, but as N increases the bandwidth available to share decreases. Bus technology suitable for small N is probably not feasible for large N, and for N somewhere in between lies a region where buses change from being effective to being unacceptable. The exact breakpoint is technology dependent and has to be evaluated for each individual type of bus and interface technology.

One possible way to build a bus with many processors is to build a physically short bus, as shown in Fig. 6.4, and to tie the processors to the bus through a longer connection that attaches to the bus through a special interface, as shown in the figure. The objective of the short bus is to provide a medium for the interchange of signals with physically acceptable parameters and good signal quality. It might be only 25 cm long, for example, and provide 100 connection points. The 100 interfaces must be located very close to the physical bus, which is possible for interfaces alone, but may be very difficult to accomplish if all 100 processors have to be physically close to the bus.

The interfaces provide signal buffering that permits the processors to be located at least far enough away to meet the packaging requirements of the processor technology. Although Fig. 6.4 suggests that the electrical bus is external to the modules that hold processors, the structure in the figure also holds to some extent for super-VLSI systems with the bus and multiple processors implemented together, possibly on a whole wafer if not on one chip.

6.3.2 Ring Interconnections

Although a bus interconnection has advantages for a small number of processors, electrical buses are highly constrained by fundamental physical principles. The goal of the architect is to find an interconnection that has the simplicity of the bus for support of computation, but is able to exceed the physical limitations inherent in buses. One possible solution is to build a logical bus that is physically something else.

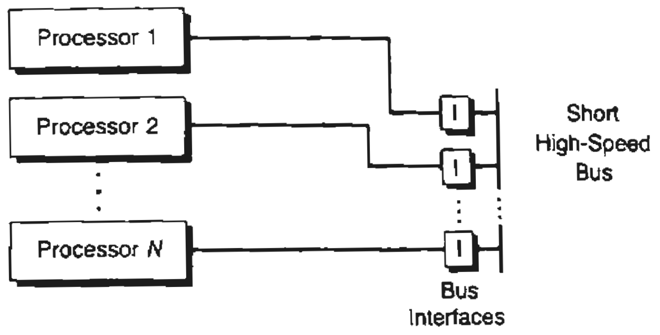

Fig. 6.4 A high-speed bus with a short physical length connecting a collection of processors. The I-unit is an interface that permits processors to be relatively far from the bus when compared to the physical length of the bus itself.

Figure 6.5 shows a loop arrangement with point-to-point connections between processors and a cyclic interconnection overall. In this system, a transmitting process places a message on the loop, and it is repeated by each receiver until it returns to the transmitter, which stops the message by failing to repeat it.

There are various ways to operate such a loop, but one protocol that turns the loop into a logical bus is the IEEE 802.5 token-ring standard. A transmitting processor is distinguished from all other processors because it holds a token, of which one and only one exists among all processors. When the transmitting processor sends a message through the token ring, the ring acts like a bus, and all other processors listen.

At the end of transmission, the transmitter broadcasts a token, which is a unique combination of signals that cannot exist in an ordinary message. Each receiver sees the token in turn, and if a receiver is waiting to be a transmitter, it accepts the token without retransmitting it, and instead transmits its message on the ring. If no receiver is waiting to transmit, the token circulates on the ring and can subsequently be removed by any processor that needs to transmit.

The advantage of the token ring is that the connections are point to point, not bus connections. Physical parameters can be more readily kept in control. In fact, the token ring is ideally suited to very high bandwidth optical fibers, which are difficult to adapt to bus technology for small numbers of processors and have not yet been adapted to buses for large numbers of processors.

A major disadvantage of the token ring is that each bus interface adds a short delay, usually a 1-bit delay, when it repeats an incoming message. As the number of processors increases, the delay around the ring increases proportionately. The bandwidth, however, does not necessarily decrease as it does for buses when they are heavily loaded.

To take advantage of the token ring, the architect views the token ring as

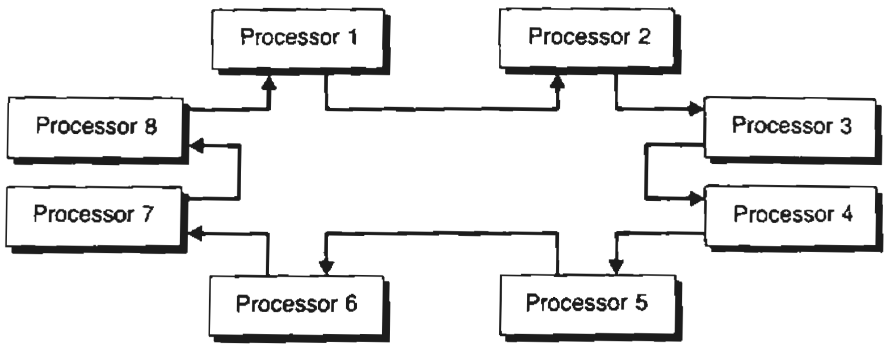

Fig. 6.5 A multiprocessor based on a loop interconnection.

if it were a pipeline with a short cycle time and long delay. The effective bandwidth can be utilized as long as computations keep the pipeline filled. Therefore, each processor should overlap transmissions with local computations.

Moreover, a protocol for a high-speed ring network ought to provide a means for a transmitter to pass its token to a new transmitter without having to wait to receive its own transmission. Such a protocol provides for pipelining messages on long rings, which is necessary to tap the available bandwidth. If a new message can be started only if no other message is on the ring, the net effect is the same as requiring a pipeline to drain between operations, which causes severe bandwidth degradation as the number of processors on the ring increases.

In today's technology, short electrical buses are limited to run at 100 to 200 MHz, depending on their length and maximum loading. Obviously, the longer and more heavily loaded buses run at the low end of the speed spectrum. Buses that are limited to the confines of a single VLSI chip can run in the high end of the range, and it is conceivable to run such systems at clock rates in excess of 200 MHz. However, if a bus leaves a chip, then maximum clock rates fall back to the 100-to-200 MHz area, and only denser packaging with special attention to low capacitance and inductance can increase the speed.

Optical connections for a token ring can run at much higher speeds. Early commercial installations of optical loops had bandwidths of 100 MHz in 1982, and by the beginning of the 1990s links running at a clock speed of 400 MHz were in use commercially. Clock rates exceeding 1 GHz are likely to appear in the mid-1990s.

6.3.3 Crossbar Interconnections

The bus interconnection offers the simplest topology but has the highest potential contention. The crossbar is the antithesis of the bus. It offers the least contention, but has the highest complexity. We take a brief look at crossbars here. In the next section we look at interconnections that fall between crossbars and buses.

Figure 6.6 shows a crossbar that connects *N* processors with *N* memories. Although the number of memories is equal to the number of processors in the figure, this need not be the case in general. Usually, the number of memories is at least equal to or a small multiple of the number of processors.

The path between a processor and memory has a delay only at the crosspoint, so each processor is a unit (one crosspoint) delay from any memory. The communications network has no contention. Contention exists only at processors and memories—that is, if Processor 1 has to access Memory 1, and Processor 2 has to access Memory 2, then both accesses can occur simultaneously in the crossbar switch. In fact, any number of simultaneous accesses up to *N* can be done simultaneously, providing that no two accesses involve the same memory or processor.

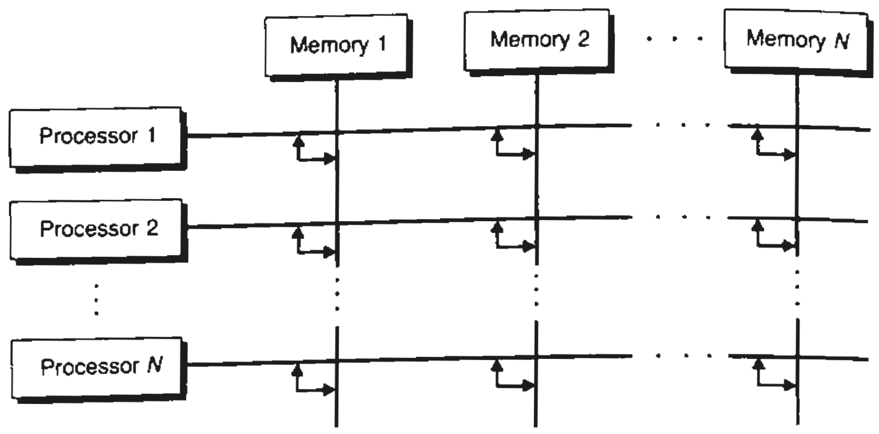

Fig. 6.6 An $N \times N$ crossbar switch in an N-processor multiprocessor. At each crossing in the network is a switch that permits any processor to connect to any memory.

Contention occurs if two or more accesses are made to the same memory. Consequently, if both Processor 1 and 2 attempt to access Memory 1 in the same cycle, one of the processors has to wait for the other to complete.

There are various architectural tricks available to reduce contention. If the contention occurs because processors are attempting to access different data that happen to be stored in the same memory module, then one possible solution is to allocate data so that accesses tend to be more evenly distributed across all memories rather than clustered to a single memory.

An obvious way to achieve this goal is to allocate blocks of data so that successive elements lie in successive modules. Similarly, shared program code should be allocated so that sequentially increasing addresses lie in successive modules. In either case, when shared data or code is accessed by two or more processors simultaneously, contention will delay one processor, and thereafter the later processor will trail the earlier processor without conflict as long as the two processors continue to access memories sequentially. This same addressing technique is used in pipelined processors that access vectors of data with a stride of unity.

If the accesses that cause contention are to a single cell or to a few shared cells, there is a more fundamental problem that requires a different approach. Some of the issues are explained in more detail in Chapter 7, but the discussion here illustrates the problem more clearly.

Consider Program 6.1, which shows the code for a processor that is forming the sum of local data and then adding the local sum to a global sum. Presumably, the local data are placed in a memory that is physically close to a processor and

Program 6.1 The use of locking to assure correct updating of a shared variable.

```
Procedure Add_to_Sum(var Global_Sum: Real, Shared;
    Global_Sum_Lock: Lock, Shared; Local_Table: array of Real);
var
  i: integer;
  Local_Sum: real;
begin
  Local_Sum := 0.0;
  for i := 1 to Max do
    Local_Sum := Local_Sum + Local_Table[i];
  {* The next statement obtains exclusive access to Global_Sum by some mechanism
    built into the architecture. At any given time, only one processor can be executing
    statements in the region between LOCK and UNLOCK. *}
  LOCK(Global_Sum_Lock);
  Global_Sum := Global_Sum + Local_Sum;
  UNLOCK(Global_Sum_Lock);
end; {* Procedure Add_to_Sum *}
```

can be accessed without contention. The shared variable *Global_Sum* is to contain the sum of all elements in the data vectors. The objective is to obtain speedup by adding the local data in parallel, then tallying the local sums into *Global_Sum*. This is much like an election process, where each precinct tallies its ballots locally, then reports the results to Election Central, where precinct tallies are summed. The problem is that the tallying at the shared datum can take O(N) time, and thereby it becomes a serious bottleneck that negates the parallelism achievable.

In Program 6.1, the local operation computation tallies data into *Local_Sum*, and from there *Local_Sum* is added to *Global_Sum*. The addition into the shared variable has to be done very carefully. Therefore, we must provide a mechanism for that variable to be read and rewritten by a single processor without an intervening operation occurring.

For example, if Processor 1 has to add the value 10 to *Global_Sum*, it must obtain the current value, add 10 to the current value, then write back the new value. If several processors attempt to do the same process concurrently, the results of global tallying can be incorrect. For example, consider the following situation in which the initial value of *Global_Sum* is 0, and Processors 1 and 2 attempt to add 10 and 15, respectively, to the sum.

1. Processor 1 reads the value 0 from *Global_Sum*.
2. Processor 2 reads the value 0 from *Global_Sum*.
3. Processor 1 computes the updated value of *Global_Sum* to be 15 and writes this back to *Global_Sum*.

4. Processor 2 computes the updated value of *Global_Sum* to be 10 and writes this back to *Global_Sum*.
5. The final value of *Global_Sum* is 10.

The error in this process causes the final outcome to miss the tally of 15 computed by Processor 1. Processor 2 reads the value of *Global_Sum* to be 0, but the instantaneous residence location of *Global_Sum* in shared memory is temporarily incorrect.

The true location of *Global_Sum* has moved to Processor 1, where it is updated and then restored in shared memory. During the time that Processor 1 "owns" *Global_Sum*, access to it in shared memory must be prevented. In essence, Processor 1 should be able to read, modify, and write *Global_Sum* as a single primitive operation without any other processor accessing *Global_Sum* in the meantime. In Program 1, this is indicated by the statements LOCK(*Global_Sum_Lock*) and UNLOCK(*Global_Sum_Lock*) that surround the READ/MODIFY/WRITE operation on *Global_Sum*. The variable *Global_Sum_Lock* is a special variable that controls access to *Global_Sum*.

The LOCK statement permits a processor to pass the statement if the variable is currently unlocked. Otherwise it forces the processor to wait until the variable becomes unlocked. A typical implementation of LOCK is to use a 0 value of a variable to denote "unlocked" and a 1 value to denote "locked". A LOCK statement waits for a lock variable to become unlocked before the processor can pass. When the LOCK discovers an unlocked variable it immediately locks it, and then continues. The UNLOCK statement unlocks a variable unconditionally. The LOCK and UNLOCK statements have to be implemented in such a way that at most one processor in a multiprocessor at a time can pass a LOCK. The instant that one processor locks a variable, every other processor will discover the variable to be locked.

One possible failure mode from improper implementation or incorrect use is a situation known as *deadlock*, in which two or more processes mutually block each other from further progress. Neither process can continue until the other unlocks a variable, but since they cannot continue, they cannot reach the unlock point in a program. An erroneous implementation of a LOCK primitive can cause deadlock if it inadvertently leaves a variable in a locked state, and no processor can thereafter unlock that variable.

If a LOCK/UNLOCK is embedded in a program, such as Program 1, then no matter how the LOCK/UNLOCK is implemented, we have a potential bottleneck in a parallel processor. In computers with bus interconnections, the bottleneck is more likely to be at the bus rather than at the memory. When the bus is replaced by a crossbar, communications bottlenecks disappear, but performance is limited by the next tightest bottleneck, which is likely to be at the shared memory.

The LOCK/UNLOCK code of Program 6.1 demonstrates a realistic way that the shared memory bottleneck can arise. Of course, the major reason to move to a crossbar is to remove a critical bottleneck that causes N simultaneous bus requests to take O(N) time. The crossbar drops this time to O(1) time, but the shared-variable bottleneck is still O(N), so all the crossbar brings us is high performance in some portions of a program, with other portions of code dominating the performance and forcing the system to operate inefficiently.

These are performance-oriented arguments. We must also look at cost. The cost of a crossbar is usually proportional to the number of crosspoints, which grows as N^2, whereas the cost of a bus grows only linearly in N since cost is proportional to the number of bus interfaces. For large N, the crossbar is extremely expensive and may well dominate the entire cost of a multiprocessor. Large crossbars are feasible only if the cost per crosspoint can be held very low. The danger in building a crosspoint switch is that the bandwidth available cannot be used effectively, so the extra cost brings little benefit.

A very interesting example of a crosspoint architecture is the C.mmp computer [Mashburn 1982] built and in operation at Carnegie-Mellon University over a span that ran from the early 1970s to the early 1980s. This architecture tied 16 PDP-11/40's to 16 memories. It was never intended to be a prototype of a commercial system, but rather served as a proving ground for developing parallel applications and parallel operating systems. As such, it stimulated a substantial pool of research results that formed the foundation of the present knowledge of multiprocessor systems.

Our major thrust is high performance, but that was not the major thrust of C.mmp. If all 16 PDP-11s could be put together on one problem to obtain a 16-fold speedup, then the total speed would be much slower than the speed available on high-end uniprocessors, although a 16-way PDP-11 might provide a less expensive way to attain that type of performance than would the purchase of a single 16-times-faster machine.

One benefit that the C.mmp did provide is the access to a memory 16 times larger than was available for a single PDP-11 at that time. Since memory was relatively expensive, the C.mmp provided a way of allocating the expensive resource among several independent processors. This was a cost-effective alternative to configuring each of N machines with a fixed amount of unshared memory. The larger shared memory provided a resource pool that could be allocated dynamically to individual processors.

The C.mmp also provided a pool of processors that could be allocated flexibly and dynamically to programs. In theory, all 16 processors could be used on a single program, or, for example, one program could be assigned five processors, another program three processors, and so on, until all processors are assigned.

In practice, programs often needed fairly large chunks of memory for individual processes, so fewer than 16 processors could easily exhaust the supply of memory. Nevertheless, the C.mmp demonstrated the feasibility of multiprocessors and parallel programming on various types of problems. This demonstration held even though the crossbar interconnection itself may not necessarily be feasible for large numbers of processors.

One can easily substitute any other connection of sufficient bandwidth for the crossbar in C.mmp, and there would be virtually no difference in performance from the crossbar-based C.mmp. The important point is that the replacement interconnection structure should be fast enough to meet the C.mmp demands without introducing a new bottleneck into the system. The new structure does not necessarily have to have a bandwidth equal to a crossbar.

C.mmp illustrates an important principle for the architect of a multiprocessor system. The total system cost and performance is the factor of major importance; the interconnection network is but one component of the system. The lesson is that if the architect expends extra effort to remove a communication bottleneck, that effort may just move the bottleneck to a different part of the system, and the cost may not be justifiable.

In terms of applications, it is most important to determine if an application can run effectively on a multiprocessor even if the communications subsystem has infinite bandwidth and is contention-free. If this can be done, then the next most important consideration is how to provide at reasonable cost a communications network whose finite bandwidth does not reduce performance below a reasonable threshold.

6.3.4 Two- and Three-Dimensional Meshes

Our discussion of architectures for the continuum model in Chapters 4 and 5 indicated that mesh interconnections have excellent characteristics for numerical problems that arise in scientific contexts. The combination of low cost and high speed for near-neighbor interactions makes such connections quite attractive for implementation.

Apart from their advantages, the most serious disadvantage is that they do not support global communication and synchronization directly. The overall speed of a parallel calculation on a mesh-based structure will depend on the proportion of global operations that have to be performed. If a mesh structure is supported by a second interconnection structure for global operations, the two structures together can provide a computer system that is well suited to a broad class of scientific applications.

Hoshino's PAX computer [1989], for example, incorporates a global synchronization bus, a global broadcast bus, and a two-dimensional mesh that connects near neighbors. Even though this architecture does not have the capacity of a crossbar network with respect to simultaneous communications between arbitrary pairs of processors, it supports a sufficiently broad spectrum of the frequently used types of communication to be effective for scientific problems. Even within this class of applications, there are instances that saturate PAX interconnections momentarily. However, the degradation due to saturation of the interconnection bandwidth can be made quite small for many scientific applications. If each processor is assigned a contiguous square region of a mesh of points on a lattice, then the calculations performed tend to grow as the area of the region whereas the communication and overhead tends to grow as the perimeter of the region. So, by assigning a suitably large region of a mesh of data points to each processor, *R/C* can be made as large as desired, and thereby decrease the relative cost of communications that are not directly supported by near-neighbor mesh connections.

In general, the longest distance travel between two arbitrary nodes in a two-dimensional mesh with *N* processors is $O(\sqrt{N})$. For 1024 processors, the worst-case delay is 32 if the end connections of the mesh are cyclic as each node is within 16 nodes of every other node on its own row and column. By a combination of row and column moves, a datum can move from any processor to any other processor. The longest path in a shuffle-exchange network grows only as $O(\log N)$, which appears to be much shorter. However, for 1024 processors, the path length is 10 stages. So the difference in path length for a mesh and shuffle-exchange network that connect 1024 processors is only a factor of 3. (The nodes themselves may change the total factor because the delays at the nodes in the two types of networks may be different.) Consequently, for multiprocessors with up to 1024 processors, performance degradation due to long paths will not be much different in a two-dimensional mesh connection as compared to a shuffle-exchange network, especially if long paths are rarely used in an application.

6.3.5 The Shuffle-Exchange Interconnection and the Combining Switch

The shuffle-exchange connection described earlier in this text can be used to interconnect independent multiple processors as well as vector processors, such as those used for cyclic reduction or recursive doubling. In this section we consider the shuffle-exchange as an alternative to the shared bus or the crossbar, since both the bandwidth and cost of the shuffle-exchange lie between those of the bus and the crossbar.

The shuffle-exchange network offers an important additional function known as a *combining switch*, which can reduce contention by performing operations *in parallel* within the network that otherwise must be serialized at the memory. This technique has excellent potential for parallel applications that require processes to have momentary exclusive access to a shared variable.

The exclusive-access requirement limits the performance of most multiprocessor architectures, so when access to a shared variable is saturated, no additional speed improvement is possible no matter how many more processors are added to the system. However, this limitation does not exist in the original designs of the RP3 and Ultracomputer systems, described later in this section, when the exclusive access can be accomplished in part in the communication network and in part in the memory. In effect, the exclusive access is done in parallel, rather than serially, by making use of facilities built into the shuffle-exchange network.

The conditions under which exclusive access can be supported efficiently by the network are rather stringent. For some applications, the combining switch satisfies the needs for serialization, but for others it might not. For those applications for which the combining switch is not suitable, either some other mechanism has to be brought into play or such applications may simply not be candidates for parallel execution except possibly on multiprocessors with a small number of processors.

The shuffle-exchange network depicted in Fig. 6.7 shows processors at one side and memories at the other. Although the memories are quite far from the processors in terms of delay, the processors can have large caches and local memories to reduce the traffic to remote memories.

The important aspect of the architecture shown in the figure is that it supports the same multiprocessor applications as do the bus and crossbar interconnections. Its bandwidth is higher than the bus, but lower than the crossbar. Its cost is $O(N \log N)$ as opposed to $O(N)$ for the bus and $O(N^2)$ for the crossbar. The shuffle-exchange network lies at an intermediate point in the spectrum of possible networks.

The bandwidth for shuffle-exchange is very high for operations that do not conflict. Lawrie [1975] has shown that if N processors place simultaneous synchronized requests so that Processor i requests data from Memory $i + c$, for any constant c, the requests can be honored simultaneously without conflict. More-

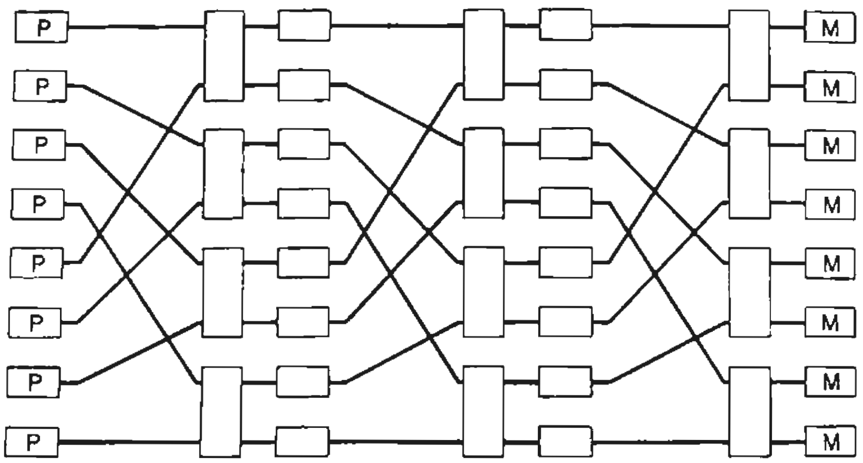

Fig. 6.7 A shuffle-exchange network for connecting eight processors to eight memories. Processors are labeled with P and memories with M.

over, no contention occurs if Processor i requests data from Memory $pi + c$, when p is an odd number and N is a power of 2.

Although we presume that the processors are independent and need not be synchronized precisely, many applications require processors to synchronize at certain points before proceeding. In most multiprocessor implementations of the fast Fourier transform (FFT), for example, each of the log N iterations is completed by all processors before the next is begun, so there are synchronization points at the end of each iteration.

Once processors are synchronized, they launch their new accesses to memory more or less concurrently. If in a vector architecture a collection of accesses to a vector has little or no contention, the equivalent accesses will tend to have low contention after synchronization in a multiprocessor architecture.

6.3.6 The Butterfly Operation and the Reverse-Binary Transformation

For the FFT there are two types of processor-to-processor communications. One is a butterfly operation, in which pairs of processors exchange data and compute weighted sums and differences of the items exchanged. The other is a reverse-binary transformation that alters the order of the output data from the ordering produced by the computations to one that is lexically ascending in the independent variable.

Cvetanovic [1987] showed that the two operations are incompatible with the shuffle-exchange operation in the sense that if data are stored among processors so that the butterfly operation proceeds without conflict, then the reverse-binary operation results in a maximum conflict in the network. Conversely, if the reverse binary is conflict free, then the butterfly results in maximum conflicts.

At least one of the two types of operations will cause some problems in the network. A typical implementation of the FFT uses log N butterfly operations on N-vectors, followed by or preceded by one reverse-binary operation. Consequently, it is best to organize data across the memories so that the butterfly is conflict free and then pay the conflict penalty for the reverse binary operation.

How bad can the conflicts be? The worst possible case is that all N items to be accessed reside in a single memory at one node of the shuffle-exchange network. O(N) time is required to obtain the data, as opposed to O(1) time if data are ideally stored across the network. However, the conflicts that arise for the reverse-binary permutation while doing the FFT are not this bad. Since the butterfly operation is assumed to be able to access N distinct items in a single operation, those items must be distributed across all memories.

When these same N items are subsequently accessed for a reverse-binary transformation, contention does not occur at the memories, but rather it occurs within the communications network. According to Cvetanovic's results, the worst-case contention for the reverse-binary permutation actually occupies only

$O(N^{1/2})$ time, not $O(N)$ time, which essentially wastes $O(N^{1/2})$ of the $O(N)$ bandwidth available.

For a permutation of data to be free of conflicts as it passes through a shuffle-exchange network, at each switch node the two operands at the inputs must be directed to two distinct outputs. A conflict occurs if the two operands go to the same destination.

The bottleneck of the network for a permutation access is the stage (or pair of stages) in the center of the network. To see why this is true, consider a permutation that has the maximum possible contention. At the first stage, the worst possible situation is for each of the $N/2$ switch nodes to direct both their inputs to only one output. This creates a situation at the second stage in which half of the inputs are empty and half have two operands.

The same contention problem can occur at each successive stage up to the middle of the network, creating $2^{(\log N)/2}$ operands queued on each of $2^{(\log N)/2}$ lines, and with all other lines empty. However, since the operands lie in distinct memories at the far end of the network, the paths followed by the queued operands in reaching the far end of the network must diverge, starting at the bottleneck. Therefore, at each successive stage the queue lengths diminish by a factor of 2, and twice as many lines become active, until at the far end all lines are active and contain one operand. Figure 6.8 shows the reverse-binary transformation for a network with 16 processors and 16 memories. For this permutation, the target of Processor i is Memory i', where i' is the integer obtained by reversing the binary digits of i. Thus Processor 2 targets Memory 4 because the reversal of $(0,0,1,0) = 2$ is $(0,1,0,0) = 4$.

The discussion on contention within the shuffle-exchange network reveals that there exist algorithms for which we must suffer $O(2^{(\log N)/2}) = O(N^{1/2})$ delay because of communication contention, even when there is no contention at the memory at all. In a crossbar network, the FFT has neither communication nor memory contention, and therefore it is potentially faster by a factor of $O(N^{1/2})$. The problem is restricted solely to the reverse-binary transformation applied at the last step, and this step is rarely discussed in the literature in evaluating parallel execution of the FFT. Cvetanovic's work has brought the communication-contention issue directly into focus.

Now that we understand the poor performance of the reverse-binary transformation, we can reduce its effects. For example, in some applications, the processing steps are:

1. Use the FFT to transform from the time domain to the frequency domain.
2. Process in the frequency domain.
3. Use the FFT to transform from the frequency domain back to the time domain.

We need not apply the reverse-binary transformation at the end of the first step if the frequency-domain operations are ordered compatibly. When the second

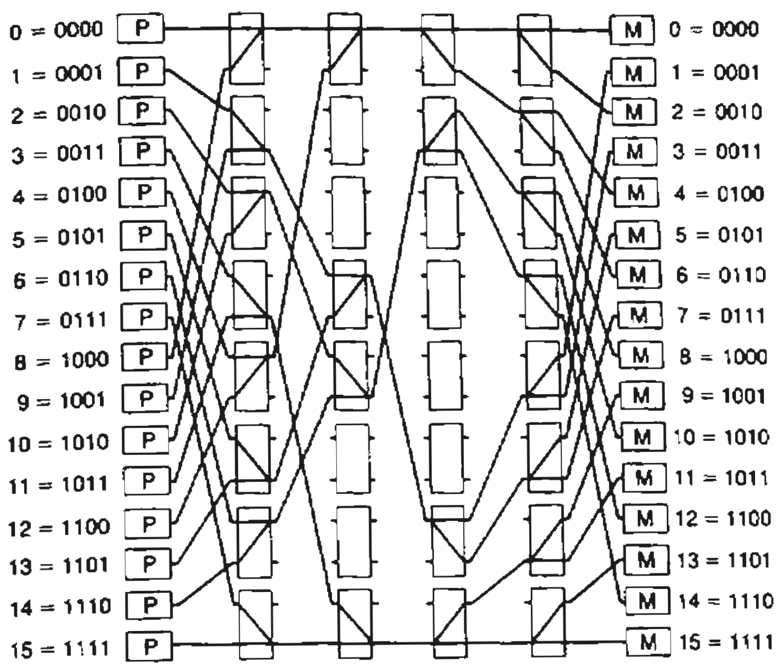

Fig. 6.8 The interconnections used to create a reverse-binary transformation in a shuffle-exchange network. Note that only some of the interconnections are used among the internal paths of the network.

step receives inputs in reverse-binary order it produces outputs in the same order. This places the input to the last step in reverse-binary order, rather than lexical order. For such an input, the FFT without a reverse-binary operation produces an output that is in lexical order. Hence, no reverse-binary transformation is performed in either the first or the third step, and the bottleneck is neatly sidestepped.

More generally, it is necessary to locate the contention problems in the communication network and to take steps to remove the problems if this is possible. The FFT is an example in which the bottleneck can be removed in the context given. We cannot promise that this is always possible, but clearly the bottlenecks have to be discovered if they are to be removed.

The discussion thus far illustrates a potential shortcoming of the shuffle-exchange network. This particular defect occurs for accesses that are balanced across the outputs of the network. But accesses do not have to be balanced at the outputs. Algorithms might well bias their accesses to memory, so that on the whole the accesses are uniformly distributed, but some small fraction of accesses is directed to a particular memory module. This might be the case if processors operate on data scattered across all memories, then reference shared control-variables to synchronize activity with other processors. We are interested in the effective bandwidth of the switch under these circumstances.

The calculation of effective bandwidth is difficult even for simpler problems. Consider the least-restrictive set of assumptions, namely that accesses are uniformly distributed and uncorrelated. The reason that this becomes difficult to evaluate is that we do not have a good model of how to deal with internal conflicts in the network. When two operands collide somewhere, for example because they both request the same output of a particular switching node, what happens? The network can

1. Abandon one arbitrarily and pass the other;
2. Queue one request in a local memory and pass the other; and
3. Refuse one request while passing the other, under the assumption that the request refused is buffered by the sender and will be repeated.

This list of options is representative but not exhaustive in the assumptions that have been treated in the literature in papers by Dias and Jump [1981], Thanawastien and Nelson [1981], Chen *et al.* [1981], Kruskal and Snir [1983], Yew *et al.* [1983], and Padmanabhan and Lawrie [1985].

Kruskal and Snir have a very elegant result based on the solution of a difference equation that describes the number of messages remaining after conflicting messages are discarded. They found that the effective bandwidth is O(N/log N), so the contention within the network reduces bandwidth by a factor of O(log N). The other researchers have obtained roughly comparable findings using queueing analyses and simulations.

The analyses in general do not relate the assumed input to the access patterns of real programs. To what extent is the literature realistic? From Cvetanovic's work on the FFT we know that the effect of periodic synchronization could be either beneficial or disastrous. Synchronization tends to cause accesses to the network to come in clumps. This is beneficial if the accesses are nonconflicting, so that a large number of accesses can be honored in a short time. It is disastrous when the accesses are highly conflicting because it causes much higher contention than predicted by statistical methods.

The architect cannot take for granted that average bandwidth will be O(N), O(N/log N), O($N^{1/2}$), O(1), or any other function that we have ascribed to the switching network. The architect has to explore the performance of the network on realistic applications, if they are available, or on faithful models of the access patterns of real applications.

This is the problem attacked by Pfister and Norton [1985] in their influential paper on hot-spot contention in shuffle-exchange networks. They sought the effective bandwidth of shuffle-exchange networks when accesses are not entirely uniformly distributed across memory. Their model permits a small number of accesses to be made to a specific memory and all others to be uniformly distributed. Their results show that effective bandwidth falls off dramatically as correlation of accesses increases.

In the Pfister-Norton model, a "hot" memory module is referenced with probability h; otherwise accesses are uniformly distributed. Therefore, when each of N processors produces r references per cycle to the memory system, the hot memory module receives requests at the rate:

$$\text{Requests at hot memory} = r(1 - h) + rhN \qquad (6.23)$$

The first term accounts for the uniform share of the load, and the second term accounts for the hot module receiving more than its share of requests from all processors.

Since a memory cannot honor more than one request per cycle, the request rate on the left-hand side of Eq. (6.23) cannot exceed unity. Therefore the maximum effective rate of generating requests, R, is the rate at which Eq. (6.23) reaches unity and is given by:

$$\text{Maximum generation rate } R = \frac{1}{1 + h(N - 1)} \qquad (6.24)$$

This function falls off dramatically with increasing N. The effective bandwidth of the switching network is N times the generation rate given in Eq. (6.24).

When h is 0, Eq. (6.24) is unity, bandwidth is N, and no degradation due to nonuniform access is present. As h increases just a little bit, for example to 1 percent, then for 1024 processors the denominator of (6.24) increases to 11, and bandwidth is down by a factor of 11 from the ideal. Even when hot-spot probability is tiny, for example 0.1 percent, the impact is an increase in the denominator to a value of 2, which reduces bandwidth by a factor of 2.

Pfister and Norton confirmed their findings by means of simulations, which showed that contention caused the network to saturate in tree-like regions, as shown in Fig. 6.9. This figure assumes that requests are held until they can be honored. The internal queue at a node can be of any integral length, including 0.

The hot memory cannot accept new data, so its predecessors become backed up when those predecessors cannot output their data to the memory. Next, the predecessors of predecessors saturate, and so on. As nodes saturate, they interfere with communication to other nodes in the system, and performance diminishes rapidly. In Fig. 6.9 the saturated nodes are indicated by shading, and they form a tree whose root is the hot memory.

A path from a processor to a different memory that has to use a saturated path becomes blocked, so bandwidth is somewhat lower than predicted by Eq. (6.24), depending on the size of the tree of saturated nodes. This in turn depends on the amount of queueing available within each node. If the architect wants to install queues in the network, Fig. 6.9 suggests that to reduce hot-spot contention, the best place to put such queues is in the rank of switches closest to the memory system. The queues might well be placed elsewhere, perhaps uniformly through the switching network to make all switches alike, to alleviate other forms of contention.

6.3.7 The Combining Network and Fetch-and-Add

Whether queues are added at the hot memory or somewhere within the network, they smooth out the effects of peak loads over longer periods. Queues do not alleviate the bottleneck caused by frequent memory accesses. To solve the problem, the request rate to the hot memory has to be decreased.

Gottlieb *et al.* [1983] propose a very unusual solution that involves using logic within the switch nodes to perform computations whose effect is to reduce the rate of requests to a shared-memory cell. In essence, two or more requests for access to the same shared cell can be combined into a single access under certain conditions. This tends to reduce the peak access-rate to a shared cell and thereby reduces contention and the bandwidth reduction due to contention.

The architectural solution is sometimes called a *combining network*, and the functional capability it gives programs is a collection of new instructions, one of which is called the Fetch-and-Add instruction.

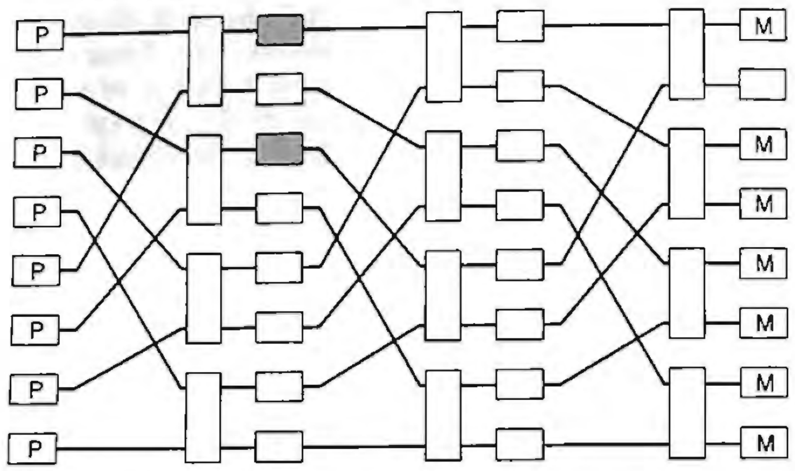

Fig. 6.9 A "hot" spot in a memory module (indicated by shading) and the switching modules that block as a result. The path from Processor 0 (the top processor) to Memory 3 is blocked, although neither Processor 0 nor Memory 3 is very active.

To illustrate how the combining switch works, we propose to examine some subtree of the communication network, namely the tree of shaded nodes that appears in Fig. 6.9, and note that its root is a specific memory module that receives more than its share of references. In this example we give a possible case for the contention and show how the Fetch-and-Add instruction solves the problem.

The sample problem is a queueing problem in which each of N requestors attempts to add an item to a queue. In conventional solutions, the queue pointers cannot be updated by two or more processors concurrently because, if this is attempted, a pointer update might be done incorrectly for the same reasons that cause a concurrent summation on a shared variable to fail. Our solution in Program 6.1 forces the updates to be done sequentially, with each process using LOCK and UNLOCK operations to obtain exclusive access to a shared variable while updating that variable.

Our present solution permits all processors or any subset of processors to update the queue pointer simultaneously. To do so, we make use of Fetch-and-Add as defined here for a single processor.

```
Definition: Fetch-and-Add (Address, Increment);

Temp := Memory[Address];
Memory[Address] := Memory Address + Increment;
Return Temp;
```

When Fetch-and-Add is used concurrently by M processors, we require the following conditions:

1. The cell at Memory[Address] is read only once and written only once, rather than read and written M times, to satisfy the M concurrent requests.
2. The set of M values returned to the M requestors is the same as some set of values that would be returned to the M requestors for some ordering of the requests executed serially with each request having exclusive access to Memory[Address] during the update of the cell.

The definition is not particularly unusual. Fetch-and-Add acts much like an Add-to-Memory instruction. The only difference of note is that Fetch-and-Add returns the prior contents of memory. The first characteristic of concurrent execution is crucial, for it is this characteristic that reduces contention in multiprocessors.

As an example of the basic idea, consider three processors that execute Fetch-and-Add concurrently to the same memory cell, SUM. If the initial value of SUM is 10, the three increments are respectively 2, 5, and 12. Then the network produces the total of the increments, 19, which is the only number added to SUM. SUM is fetched once to obtain the value 10, and the new value 29 = 19 + 10 is the updated value of SUM. Meanwhile the network computes the values to return to the three requestors. One possible set of values that could be returned is 10, 12, and 17, which are the values that would have been returned had the increments 2, 5, and 12 been used sequentially in that order.

The trick to the implementation is illustrated in Fig. 6.10, where we see how the cells in the shaded subtree produce the necessary behavior. Each cell combines data moving toward memory and does an inverse operation for data moving away from memory. In this case, each cell detects when two Fetch-and-Add operations for the same shared variable reach its inputs simultaneously. The two increments are added internally to produce a sum, which is routed to the memory. Thus, one cell adds 2 and 5 to produce 7, and the second cell adds 7 and 12 to produce 19.

To prepare for the return trip, each cell stores the value of one of the two increments, in this case the left-hand input. Hence the first cell stores the value 2, and the second one stores the value 7. By storing the value of the left-hand input, when data traverse the network from memory to processors, the results returned will be as if the left-hand increment were used before the right-hand increment to update the shared variable. In this case, on the return trip, the number 10 reaches the cell with the stored value. It places the 10 on the left-hand port, and the sum 17 = 10 + 7 on the right-hand port. The right-hand port now has a value that would be seen if the value of SUM were 17 just before the 12 were added to it.

Meanwhile the value 10 travels to the first cell. There the unmodified value

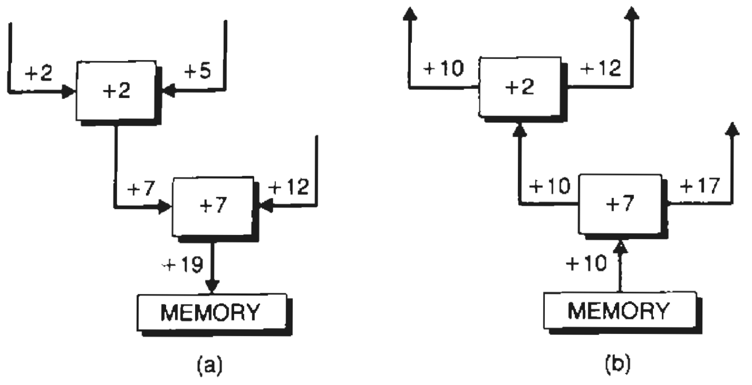

Fig. 6.10 Two phases of a Fetch-and-Add instruction:
(a) The data flow towards memory when increments of 2, 5, and 12 are applied. The numbers in the switch cells show the saved datum; and
(b) The data flow away from memory for the return of information to the requesting processors. The memory returns the value + 10, and the switching cells modify the returned datum as shown before reporting the datum back to the requestor.

of 10 is reported to the left port, and the sum 12 = 10 + 2 is reported to the right port. The left port, therefore, has a value of 10, which would be the value before the increment 2 is used to update SUM. The right-hand port has the value 12, which is the value it would see if SUM were updated by 2 just before the 5 from the right-hand port is used to update SUM.

Each cell in the combining switch has at least the following capabilities:

1. Detect a matching address on left and right inputs.
2. Add two increments.
3. Save one increment.
4. Match a returning value for Fetch-and-Add to a saved increment for the instruction.

These capabilities in a combining switch are fairly costly, but the combining switch potentially can increase performance due to hot-spot contention by removing critical sections for some shared variables. An open question is whether the cost of the combining network is justified by its impact on performance.

As a concrete example of an extremely important use of Fetch-and-Add, consider the problem of enqueueing and dequeueing requests in a multiprocessor. An obvious mechanism for controlling a multiprocessor is to place tasks on a queue when no processor is available to execute them. As a processor completes its present work, it inspects the queue and removes a new task for execution if there is one.

The queue itself is a bottleneck when queue pointers must be locked and unlocked for safe updating. If, for example, a queue holds N independent tasks, all ready for immediate execution, and N processors suddenly complete a phase of activity and become available for new task assignments, ideally we would like to hand over the tasks in a single cycle so that all processors can start immediately. However, when pointer updating is serialized, then handing out the tasks takes $O(N)$ time, which could be quite significant for large N. This overhead is intolerable if the tasks are short, for example $O(1)$ time in length.

The basic idea in using the Fetch-and-Add is that each processor attempting to enqueue an item requests a position in the queue. This can be done with a statement of the form:

```
enqueue_position := Fetch_and_Add(Head,1);
```

In this case the first argument of Fetch-and-Add is a counter, Head, which gives the present position in the queue at which an item is to be added. The second argument is the increment by which Head is increased when a new item is added to the queue.

When the code is executed serially, the Fetch-and-Add returns the position of the next item. When the code is executed concurrently by two or more processors, all Fetch-and-Adds can be done at the same time, yet each processor will receive a unique, valid index into the queue because the values returned by Fetch-and-Add are the same values that would have been returned for some serialization of the Fetch-and-Adds. Any serialization of the enqueue requests yields correct code for sequencing *N* requests, and the Fetch-and-Add mimics one such serialization, but it does so with as little as one memory cycle.

We have not treated here the need to make the queue cyclical, nor have we treated the case of the empty or full queue. Chapter 7 studies these programming issues more fully. The example has served our purposes sufficiently well to show the potential use of the Fetch-and-Add instruction. In spite of the potential improvement offered by Fetch-and-Add, it is uncertain whether it is worthwhile incorporating into a multiprocessor, and if so, to determine how many processors must be in the system in order to gain sufficient performance improvement to offset the cost of the implementation.

In the ideal case, the combining network removes a bottleneck, and the next bottleneck is at a much higher level of throughput. The value of the combining network is the gain in speed in being able to operate at a much higher throughput rate than permitted without the combining network. However, it is quite possible to find that the combining network eliminates a bottleneck that is only marginally below the next bottleneck in the system, so its cost is hardly justified in such circumstances.

An essential element of the Fetch-and-Add instruction is that it returns data sufficient to serialize a computation. Sullivan *et al.* [1977] proposed a machine that reduces bandwidth by combining read accesses to a common address in memory. If two or more accesses ask for the same item, the shuffle-exchange network in their architecture has the ability to combine the multiple requests into a single request and route the resulting data from memory to all requestors. Thus, the proposal by Sullivan *et al.* illustrates how to embed a broadcast-like capability into the shuffle-exchange network to combine multiple read accesses, and this capability is retained in the Fetch-and-Add implementation. This design undoubtedly influenced the inventors of the Fetch-and-Add, but it is generally less useful than is the Fetch-and-Add because of Fetch-and-Add's additional ability to perform arithmetic as part of the combining process. It is this additional ability that gives Fetch-and-Add the potential for eliminating hot spots due to synchronization and queueing traffic.

Can Fetch-and-Add eliminate hot-spot contention as actually observed in practical applications? A hot memory can be hot if it receives a disproportionate number of accesses for any reason, but a combining network is effective only if all those accesses are to the same address. Is this case realistic? Yes, it is if the reason for the biased distribution of accesses is due to accesses to shared data such as for synchronization and locking. If the hot-spot contention is for other reasons, then Fetch-and-Add is of minimal benefit. What is the answer?

A research computer called the *RP3* explored this question and other related ones at IBM in the late 1980s [Pfister *et al.* 1985]. Its structure is outlined in Fig.

6.11. At the left is a processor, one of 64 in the operational configuration, and at the right is a network of shuffle-exchange stages. The original design of RP3 incorporated two distinct networks between processors and memories—one a conventional shuffle-exchange network designed for low latency and high-bandwidth, and the other a combining network that supports the Fetch-and-Add. The idea of using two networks is that the noncombinable accesses should be directed to the fast, conventional network, and that the combinable accesses produced by Fetch-and-Add should be routed through the combining network. The higher latency of the combining network is charged only to the requests that might be combined, and thus the majority of the requests are not affected by the additional latency.

As RP3's design evolved, the combining network was dropped from the implementation because the development cost was disproportionately high for the potential performance improvement in a 64-processor system. The Ultracomputer project at NYU also dropped its planned implementation of Fetch-and-Add. Consequently, no implementation of a combining net is in progress at the time of the writing of this text. Thus, the finding so far is that the cost of hardware combining is high enough to deter its use.

Nevertheless, let us return to the description of the shuffle-exchange network in the RP3, and in particular, to look at an interesting idea embodied in the implementation. The network in Fig. 6.11 is shown with its inputs and outputs on the same side. In effect each processor node of Fig. 6.9 is identical to the corresponding memory node in that figure. The global memory is spread among the processors so that each processor has one independent block of memory, some of which can be used as global memory, and the remainder of which is used for local data. Between the processor and the network is an address mapper, a cache, and an interface for routing requests to local or global memory or to the network, where it can be routed to a remote block of global or local memory.

Addressing in this system is rather novel. To reduce contention, it is ex-

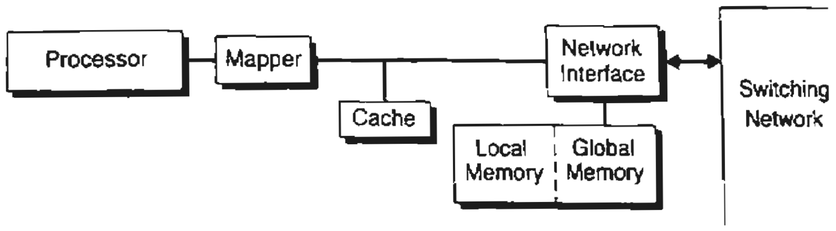

Fig. 6.11 The structure of one of 64 processors of the IBM RP3. The switching network is a shuffle-exchange network with combining logic.

tremely advantageous to distribute the global address-space evenly across all memory modules to balance requests across all modules. This is most easily done by using the least-significant bits of a memory address to specify the module that has the data. Then references to items close to each other in the logical address-space are scattered more or less uniformly to all physical modules. Local memory, however, cannot be treated in the same way. Local memory should be physically close to its associated processor. Local memory should use the most-significant, not the least-significant, bits to select a physical memory. Thus, items that lie close to each other in the address space of local memory should lie in the same physical memory module.

RP3's approach to this dilemma is to use a boundary within the address space to separate the subspace that has interleaved addresses from the subspace that has block addressing. If an effective address falls above the boundary, for example, then the least-significant bits determine the physical module, and the most-significant bits are the address within module. If an effective address falls below the boundary, the most-significant bits determine the physical module and the least-significant bits are the address within the module. In the former case, the address subspace is used for shared, global data, and in the latter case, the address subspace is used for local data.

Local data are not private in the sense that it is possible for a processor to produce an address in the local address space of a remote processor, but the main objective is to use the local address space for items that are unshared and frequently accessed and that should be held in close proximity to a processor. The RP3 has an additional degree of freedom in that the boundary between local and global subspaces is software controllable. Thus a control program can select a suitable ratio for the sizes of the subspaces, and this is not fixed in advance by the hardware.

6.3.8 Hypercube Interconnections

In our discussion of interconnections we have covered an extensive range of possibilities that illustrate the variety of trade-offs in cost and performance available in a multiprocessor. The shuffle-exchange network and the two-dimensional mesh network lie somewhere in the middle of the possible trade-offs, where buses represent one extreme and crossbars represent the other. Note that both the shuffle-exchange and the mesh connections have a small permissible fan-in and fan-out per network node. This reduces cost. The network topology determines performance and link bandwidth per wire.

The low fan-in and fan-out of a shuffle-exchange network can be increased, and thereby reduce the number of nodes on the longest paths in a network. Several hypercube computers based on this general principle were introduced in the mid-1980s, the most parallel being the Connection Machine, with 64K 1-bit processors [Hillis 1986], and the most influential being the Cosmic Cube

[Seitz 1985], that has been described in Chapter 4 in greater detail. Fox *et al.* [1988] analyze the program implementation and performance of a number of scientific applications on an Intel Hypercube, and bring together a number of important research results that pertain to hypercubes in general.

It is interesting that the Connection Machine implements combining by means of software that exploits the topology of the hypercube connections. Because the hypercube connection pattern is an extension of the shuffle-exchange connection, the notion of a combining switch for the shuffle-exchange network extends to the hypercube network by analogy. The details of the software implementation are in Hillis and Steele [1986].

In all cases, from bus to crossbar and in between, the ratio *R/C* determines how many processors can fruitfully be put to work on a single problem simultaneously. The bus has the lowest potential value of *R/C*, and it is the topology most likely to be ineffective as the number of processors increases. Note that the architecture of the RP3 attempts to keep local data and frequently used data within a processor, thereby increasing the *R/C* ratio as well as the number of processors that can be used effectively.

At this writing the multiprocessor is still in its infancy in the commercial world. One dramatic lesson of the experience obtained thus far is that the major unknown area to explore is software. What are good parallel algorithms for solving various important problems? The key approach is the ability to partition the problem into modules that require relatively little intermodule communication. If the partitioning can be done successfully, then communication requirements are rather small, and the dependency on the interconnection topology is greatly diminished. On the other hand, if communication requirements cannot be made small, then the interconnection topology becomes important, and the major parameter of interest is the *R/C* ratio.

6.4 Cache Coherence in Multiprocessors

The key to using interconnection networks in processors is to send data over the networks rather rarely. This tends to reduce contention, and, as the use per processor diminishes, the number of processors that can be served increases. Obviously, a cache memory provides an effective means for maintaining local copies of data to reduce the need to traverse a network for remote data.

We point out in the previous section that if a cache misses only 10 percent of the time, and remote fetches occur only on misses, then the number of processors supportable on the interconnection network is ten times greater than for a cacheless processor. The multiplier climbs inversely with the miss ratio, so the potential parallelism is quite dramatic when the miss ratio is near 0.

Caches in multiprocessors must operate in concert with each other. Specifically, any datum that can be updated simultaneously by two or more processors must be treated in a special way so that its value can be updated successfully regardless of the instantaneous location of the most recent version of the datum. The purpose of this section is to explore multiprocessor caches and examine the control algorithms required for these caches to behave correctly.

First, let us examine the nature of how caches might reach inconsistent states. This will give us some insight into mechanisms suitable for correcting the problem.

We have discussed the special requirement for handling shared variables in memory, and a similar requirement holds for shared variables in caches. When a shared variable is resident in memory, we can view the memory cell as being the current residence of the variable.

Earlier in this chapter we find a problem in trying to update the value of a variable shared by two processors. What goes wrong with the update process is that momentarily the current value of the shared variable moves from memory to the first processor, Processor 1. While Processor 1 holds the current value and updates that value, Processor 2 accesses shared memory. But the current value of the variable is no longer there. The variable has moved to Processor 1, yet Processor 2's request is not redirected. It erroneously goes to the normal place for storing the shared variable.

Our example presumes that Processor 1 updates the shared variable and immediately returns it to memory, but in a cache-based system, Processor 1 may well hold the variable indefinitely in the cache. The failure exhibited in the example becomes much more likely when caches are present. The failure interval is not limited to a very brief update period, but can happen for any access to the variable in shared memory while that variable is held in Processor 1's cache. Whether the failure probability is low or high, the treatment of shared variables must be handled correctly. There has to be some solution that has truly zero probability of failure. Can you imagine the havoc wreaked in a system in which this were not the case? Programs would almost always work correctly, but would fail randomly when timing conditions caused the shared variables to be misread. The failures would be nonrepeatable and extremely difficult to diagnose. They might well be misdiagnosed as intermittent hardware failures.

There is a related failure mode that also has to be considered. If Processor 1 copies a shared variable to its cache and updates that variable both in cache and in shared memory, then problems can arise if the values in cache and in shared memory do not track each other identically afterward.

Suppose, for example, that Processor 2 updates shared memory. At a later time Processor 1 requests the value of the variable, but takes that value from its copy in the cache and ignores altogether the change in the variable from the update performed by Processor 2. Processor 1's access is to a stale copy of the data held in cache, and it should be to the fresh data held in shared memory.

Another form of the stale-data problem occurs when a program's footprint is not flushed completely from cache when that program is moved to a different processor and returns at a later time. Suppose that Processor 1 is running a program that leaves in cache the value 0 for variable X. Then the program shifts to a different processor and writes a new value of 1 for variable X in the cache of that processor. Finally, the program shifts back to Processor 1 and attempts to read the current value of X. It obtains the old, stale value of 0 when it should have obtained the new, fresh value of 1 for X. Note that X does not have to be a shared variable for this type of error to occur.

In all failure modes discussed here, the common problem is for each processor to direct its memory accesses to the current active location of any variable whose true physical location can change. Simple solutions are possible, but they have performance penalties.

For example, each shared datum can be made noncacheable to eliminate the difficulty in finding its current location among *N* caches and main memory. This can be done, for example, by providing a special range of addresses for noncacheable data, or by using special LOAD and STORE instructions that do not access cache at all.

To eliminate stale-data problems for cacheable, nonshared data, the processor can flush its cache each time a program leaves a processor. This guarantees that main memory becomes the current active location for each variable formerly held in cache.

While these simple solutions have been adopted in some multiprocessors, the solutions have a negative effect on performance because they reduce the effective use of cache. We want to explore other solutions that retain a higher effective use of cache while still guaranteeing that the total system can operate error free.

The general problem is called the *cache-coherence* problem, and it has been studied in the literature by Dubois and Briggs [1982] and Archibald and Baer [1986]. These articles examine the performance impact of protocols for maintaining consistent caches. Goodman [1983] is an early paper that outlines in detail a reasonably efficient cache-coherence mechanism. Sweazey and Smith [1986] explore a variety of cache-coherence protocols and delineate virtually all the possible variations of the Goodman proposal.

Of the many proposals, our discussion focuses on a single reasonable solution to cache coherence and small variations of the basic idea. We examine its characteristics to determine its performance limitations in a multiprocessor. Architects should be familiar with the entire spectrum of protocols and with the relative performance of different solutions as measured on their own workloads on their own machine environments. We specifically do not recommend any one approach because the actual choice of the best protocol is quite dependent on the computer structure and the workload for which it is used.

To understand how to implement cache coherence, let us first describe what is required. An important notion that solves the problems mentioned is that the WRITEs to each memory location occur in a serial order, and that all processors observe the WRITEs in this order if they access the memory location. This prevents one processor from observing a sequence of WRITEs to location X to occur in the order 1, 2, 3 while a second processor sees the WRITEs occur in the order 1, 3, 2. It does, however, permit a processor to miss an observation and see the sequence 1, 3 or 2, 3. This form of cache coherence forces arbitration to take place when two or more processors attempt to write into a shared location. Only one processor can have a write privilege at a time. If that processor, say Processor A, has a write privilege and has a local copy of the variable in cache that is writeable, then the current logical location of that variable is the cache of Processor A.

This description of cache coherence has evolved over time as multiprocessors with caches have been offered commercially in various configurations. Cache-coherence mechanisms for bus-based multiprocessors automatically serialize all accesses across machines because processors have to contend for the bus and the access rights that are granted to one processor at a time. Hence, the mechanisms proposed to solve coherence problems for bus-based systems implicitly make use of the serialization inherent in gaining access to the bus. Our definition makes explicit the requirement to serialize the sequence of WRITEs to each location because, except for bus-based architectures, serialization is not inherent in the multiprocessor topology and has to be designed into the architecture. The definition of coherence used in this discussion follows the description used by Gharachorloo, *et al.* [1990]. As we learn in the next section, once we leave the safety of the serialized bus-based implementation, our troubles are not limited to coherence problems, and we have to take additional steps to assure correctness of multiprocessor programs.

Now let us return to the implementation of mechanisms to enforce cache coherence. Here are the basic operations that must take place:

1. If a READ operation for a shared datum misses in cache, then the READ operation must be redirected to the current logical residence of the variable. The variable may be in a cache, in main memory, or copies may be in many places. The READ operation should receive the most current value of the datum although by the time the copy reaches the requestor it could be stale.
2. A WRITE operation to a shared datum, whether it is a hit or a miss in cache, must have a privilege to change the value of the variable. The current logical home of the variable is at the processor that is granted write privilege, and only that processor has write privilege. Write privilege is passed sequentially from one writer to the next, and the sequential order of granting the privilege determines the serial order of values assumed by the variable.

It is convenient to keep track of write privilege in cache by associating an *ownership* bit with each cache line. The processor with write privilege to a line is deemed to be the *owner* of a line, and signifies ownership by setting the bit. All other processors that hold copies of the line have their ownership bit reset, and they must become owners before they are entitled to change the value of the line in their respective caches.

Before discussing an implementation of these requirements, note that there is a potentially severe performance penalty associated with cache-coherence protocols. The first requirement could be implemented by a broadcast operation to every processor, and every processor then performs a cache read in response. This tends to increase network contention and reduces available cache bandwidth. Since this operation takes place only on read misses, its frequency should be just a few percent of the READs issued by any single processor.

As the number of processors increases, however, the broadcast READ requests from the collection of processors create an enormous amount of communications network messages and cache traffic, so that the network or the caches or both quickly reach their saturation limit. For example, a 1 percent miss rate on shared data in each of 100 processors of a multiprocessor generates $100 \times 0.01 = 1$ broadcast request and 1 cache read per clock cycle in each processor. This much broadcast traffic saturates the communications system and the individual caches of all processors. A solution other than broadcasting of messages has to be brought into play.

A suitable alternative is to maintain a distributed directory of current holders of cache lines, and to have READ requests routed to such directories. Lenoski, *et al.*, [1990], for example, describe the details of such a scheme for the DASH multiprocessor. Another scheme is the Scalable Coherent Interface [James *et al.* 1990], which is designed to permit systems with up to 64K processors to maintain cache coherence.

Even greater potential degradation can be caused by the second requirement, if it were to be met with a broadcast on every WRITE to a shared datum while copies exist elsewhere in the system. The difference between the READ and WRITE penalties is that immediately after a READ miss occurs, the shared item becomes available in a local cache, and subsequent READs can be performed without broadcast. For WRITE operations, however, if two or more processors attempt to access and modify the same shared variable several times over a brief period of time, and if the requests by each processor are interleaved in some order, then the cache-coherence protocol generally causes heavy traffic due to frequent broadcasts that progressively move the datum from one cache to another as the privilege to write the variable transfers back and forth repeatedly. Although this behavior appears to be unlikely, it is extremely likely to occur in multiprocessor systems at barriers in programs and at locks that protect regions requiring exclusive access.

The basic mechanism for broadcast is best suited for a bus interconnection because a bus transaction is automatically assured that all receivers are listening to the bus when the transmitting processor gains access to the bus. Broadcasts can easily be implemented in shuffle-exchange networks and hypercubes, but they suffer from the problem that extra bandwidth available in these networks is lost momentarily when a broadcast saturates the interconnection network.

Similarly, a crossbar network is saturated by a single broadcast message, and that broadcast has to be delayed until all receivers are listening, which causes additional loss of useful bandwidth. Most proposals for cache-coherence protocols are therefore based on bus-connected multiprocessors. The RP3, for example, with its combining-switch network does not have a cache-coherence protocol, but instead caches only nonshareable data. References to shared data are routed directly to memory without interrogating cache.

Given the basic principles of cache coherence, the least complex solution is to broadcast a READ on every read miss of shared data, and to broadcast a WRITE on every write to shared data. This is ideally suited to a bus-based system because the bus is a broadcast medium. Goodman [1983] provided the starting point for research and evaluation of protocols for bus-based systems because of the inherent efficiency of his protocol for that environment.

In the broadcast environment, each cache listener responds to a READ by interrogating its own cache and reporting back the data. This works fine if there is only one respondent, or if all respondents have the same data. Then the first one to respond broadcasts a reply. All other potential respondents observe this and withdraw their attempts to reply.

Because there could be stale data, not all copies may be identical. To distinguish the current valid copy from all others, recall that at most one copy held by a processor has a WRITE privilege attached to it. The one with WRITE privilege is deemed to be the current copy of the datum, and the owner of this datum must be the only respondent. It may happen that no processor has a copy with WRITE privilege because the item has long since been flushed from a cache where it once resided. The default situation is that the item is returned from main memory when no processor holds a copy of the request; in a bus-based system (where all copies of a line in various cache are identical) the item is returned from any processor that holds the cache line.

Let us now look at the execution of a WRITE to shared data in an environment that supports inexpensive broadcast. When a WRITE request is received by a processor that holds the copy of the datum in local cache, the processor recognizes that the local copy is about to become stale. The processor can respond in one of two ways, depending on the details of the protocol implemented. One response is to update the local value by replacing it with the value broadcast with the WRITE request. This maintains the current value locally and assures that the caches across the system remain coherent. Protocols that use this technique are called *write-update* protocols. The other alternative is to purge the copy from local cache. This operation is called a *cache invalidation*. The protocols that use this means are called *write-invalidate* protocols. The decision whether to use write-update protocols or write-invalidate protocols depends on such factors as the cache size and the likelihood of accessing a shared variable again in the immediate future.

Note that a WRITE request can result in an operation that looks much like a READ as well as a WRITE. The response to a WRITE request is a copy of the current contents of the cache line from the owner of the line or from main memory if no processor is a current owner. The requestor usually writes only one or two words of data into the line, whereas the cache line size may contain many words. So the part of the line that is not modified by the requestor has to be sent to the requestor in order to assure that the requestor has the most up-to-date version of the cache line. Hence, this part of the transaction acts like a READ. If the protocol is a write-update protocol, the requestor must also send out the changes made by the WRITE. Listeners see the full transaction, so that any processor that participates as an updater receives sufficient information to create an updated copy of the cache line.

One advantage of maintaining an ownership bit is that it provides a potential improvement on the basic algorithm by eliminating broadcasts when they are known to be unnecessary. The idea is to eliminate broadcasts when the owner's copy is the only copy among processor caches. Then the owner is free to change the local value without telling other processors of the changes made. So this variation of the coherence protocol uses a second status bit for each cache line called the *exclusive* bit, which is set when no other copies exist in other caches. The rule for WRITE broadcasts is to broadcast a request unless the cache-line status indicates that the copy in the local cache is owned exclusively by the processor.

Most implementations of this variation of the protocol go one step further. The WRITE broadcasts are implemented as invalidates, rather than as updates. At the conclusion of such a WRITE update, the owner of the cache line can set both the exclusive bit and the ownership bit, and thereafter is free to write into the cache line without broadcasting until a broadcast READ or WRITE for that line from another processor is received. If the request received is for a READ, the local copy must reset its exclusive bit to indicate that copies exist in other processor caches. If the request is for a WRITE, the local copy has to be invalidated under the rules of the protocol so that the new requestor will be the exclusive owner of the cache line.

Although this seems to be a reasonable optimization, in Chapter 7 we show that for certain kinds of synchronization the write-update is superior. The reason that write update is superior occasionally is that the shared data ought not to be removed from a cache when the processor happens to be at a point in the code where the processor will rerequest a copy of that variable. In anticipation of the future need of the processor, the better solution would be to use a write-update protocol for that processor and place the new value of the shared data in the cache in advance of its need.

The decision between using write-invalidate or write-update can be made for individual cache lines, and different processors can select arbitrarily between the protocols for the same cache line, provided that the exclusive and ownership bits are correctly set at the end of each operation. If a WRITE request is posted with a control indicator stating that it is a write invalidate, then all listeners invalidate their copies. The requestor then can set the state of the cache line to be owned and exclusive. If the request is indicated to be a write update, the listeners are free to update or invalidate as they choose, but no listener can also be an owner of the cache line at the close of the operation.

This discussion has been directed to protocols that support broadcast messages. They are simple to understand and easy to implement on a bus topology. As multiprocessor systems grow to hundreds and thousands of processors, the bus topology becomes unusable and the excessive cost of broadcasting prohibits the implementation of protocols as described here. However, the protocols proposed and implemented in practice are logically very similar to the solutions presented here, so this discussion serves as a natural starting point to examine other implementations. The DASH multiprocessor protocol [Lenoski 1990] is an interesting place to initiate a study of coherence protocols for systems without a broadcast bus.

The important points to retain when implementing other protocols are:

1. The protocol has to identify at most one processor as the owner of the cache line.
2. No processor can write to a cache line unless the processor is the owner of the line.
3. If an owner writes to a line, the owner must notify the processors that currently hold copies of the line that the line has been changed.
4. An efficient protocol is one in which the cost of notifying other processors of changes is small and unobtrusive.

Very little is known today about the likely access patterns to shared data in multiprocessors, so all coherence protocols are worthy of consideration in the immediate future. As multiprocessors become more widely used, performance data can be used to evaluate the protocols and identify which one or ones are best for specific implementations that become available.

6.5 Memory Consistency in Multiprocessors

In a landmark paper, Dijkstra [1965] described how to control multiprocessors without the aid of synchronizing instructions. His paper was written before caches became commonplace so he did not have to worry about problems of coherence or incoherence of the memory system. The correctness of his algorithm rests on a fundamental assumption about how multiprocessors work. Namely, if a processor performs WRITE A and WRITE B in that order, then all other processors see those writes performed in the same order. Dijkstra did not make this assumption explicit in his paper, but his proof relies on this principle.

In a later paper, Lamport [1979] made this observation explicit, and argues convincingly that this basic principle must hold within multiprocessors to simplify programming them. In fact, the principle that Lamport described is somewhat stronger. Not only do WRITEs by each processor have to be observed to happen in the same order by all other processors, but the whole system must operate as if all READs and WRITEs by the various processors are merged into one sequential ordering, and each processor's operations appear in their order of execution within that ordering. Lamport calls this *sequential consistency*.

The reason for requiring sequential consistency becomes clear when you consider what Program 6.2 produces when sequential consistency is violated. Assume that the two programs in the example execute in an unknown order, and look at the final values of the variables. If the execution is sequentially consistent, the values of A and B cannot both be 0. For example, if the final value of B is 0, then Processor 1 must have written B after Processor 2 completed the execution of both instructions. So the final value of A must be 1. By similar reasoning, if the final value of A is 0, the final value of B is 1.

For many multiprocessor systems it is perfectly possible to discover that both A and B have values of zero, which seems to be precluded by the way the programs are written. The execution appears as if the WRITEs of one processor occur in reverse order. An implementation produces this result if there is more than one path between processor and memory, and if the first operation follows a different, much slower path from processor to memory than the second one follows. Although Program 6.2 is an example that does not read shared variables, Program 6.3 contains both a READ and a WRITE to shared variables A and B, and it fails to give consistent answers when sequential consistency does not

Program 6.2 An example of WRITEs by a processor that can be observed out of order.

```
Processor 1 Program
B := 0;
A := 1;.
...
Processor 2 Program
A := 0;.
B := 1;
...
```

Program 6.3 An example of a READ and WRITE by a processor that can be observed out of order. Processor 1 initializes the variables well before the READ and WRITE section occurs.

```
Processor 1 Program
A := -1;   {* Initial values of variables *}
B := -1;   {* Initial values of variables *}
...
A := 0;
D := B;.
...
Processor 2 Program
B := 0;.
C := A;
...
```

hold. A failure occurs when the interleaving of the last two statements shown for Processor 1 can be perturbed relative to the similar statements executed by Processor 2. Assume that the initialization of A and B shown in the code for Processor 1 is done well in the past. When sequential consistency holds, then as both Processors 1 and 2 complete their two update statements, the final values of C and D are both 0 or one them is -1 and the other is 0. Variable C is -1 if Processor 2 finishes both instructions before Processor 1 begins its pair, and D has the value -1 if Processor 1 finishes both instructions before Processor 2 begins its pair. Consider what happens if Processor 1 completes its pair in the order stated but Processor 2 observes the order to be reversed. A possible outcome would be an execution as if the following events occurred in sequence:

1. $D := B$ on Processor 1. D now is -1.
2. $B := 0$ on Processor 2. B now has the value 0.
3. $C := A$ on Processor 2. C now has the value -1.
4. $A := 0$ on Processor 1.

In this case, both C and D receive the values -1, which is not a possible outcome for sequentially consistent hardware.

Note that these results do not violate cache coherence because the values received by each variable in the listed sequence of actions are serialized and the processors observe the same sequence of changes. The inconsistency in this example is that the given sequence of steps is not a permitted interleaving of the pairs of statements of the two programs.

Hence the principle of sequential consistency is different from cache coherence, because cache coherence states what must hold for individual locations in memory but makes no statement about the relative order of READs and WRITEs to different locations, and the order in which they can be observed. Sequential consistency is easy to assure in a bus-based multiprocessor with a cache-coherence protocol because cache coherence assures that all observations of the sequence of values taken by each variable will be the same, and sequential consistency is assured because the shared READs and WRITEs are serialized by the bus transaction. READs and WRITEs to privately held data are not serialized by the bus, but the order in which they occur can be merged consistently into a total serial ordering of all transactions, as required by the principle of sequential consistency.

Parallel programs are sufficiently difficult that it makes little sense to add to their complexity by building hardware that does not support sequential consistency. But sequential consistency incurs a great performance penalty when implemented in multiprocessors other than bus-based systems. Sequential consistency requires that all instructions issued by each processor, say Processor 1, appear to execute in the same order when observed by all other processors. That order must be the program execution order on Processor 1. The problem is that it is usual practice to execute instructions out of order to attain performance improvement, provided that the program behaves on the local processor as if the instructions were executed in order. For example, the instruction sequence WRITE A, READ B, can be reordered to take advantage of being able to start the READ access a little earlier. The processor can continue execution immediately after a WRITE is issued, whether or not the WRITE misses. But it usually suspends execution if a READ misses, waiting for the data to be returned to a machine register where subsequent instructions can deal with them. By executing the READ early, if the READ misses while waiting for data from cache, the WRITE can be executed, and thus some useful work can be done during time normally left idle.

For this reason, high-performance machines execute some instruction sequences out of order provided that out-of-order execution does not change the program correctness in a uniprocessor environment. If the READ and WRITE instructions that are interchanged access different memory locations and manipulate different registers, then the READ and WRITE can be interchanged. So the hardware freely makes such changes to obtain performance improvements. The hardware need not actually interchange the order of execution but may have WRITE buffers in one or more places so that events are made visible to other processors as if they occurred in a different order.

Sequential consistency may force processors to exchange messages in order to assure that global timing of events remains sequentially consistent. For example, we mentioned that a sequence of WRITEs can appear to have been reversed if the first WRITE follows a long path to memory and the second one follows a short path. To assure that the WRITEs are treated in a correct order, it may be necessary to wait for the first to complete before launching the second.

This implies that the memory returns a message to the processor when a WRITE is completed.

What we have learned is that the hardware designer and the software developer have to agree on a way to assure correctness of parallel programs that is compatible with high-performance design. The trend today is to build systems with a special subset of instructions designated for the purpose of synchronizing events across machines. The ordinary instructions execute as fast as hardware permits. They may in themselves appear to be sequentially inconsistent, but the special instructions assure that where timing consistency of some sort is necessary for correctness, the consistency among the special instructions will assure correctness of the full program.

Program 6.1 gives an example of how to write correct programs under these conditions. Assume that the primitive instructions that implement LOCK and UNLOCK are the special instructions, and the instructions inside the critical sections are ordinary instructions. For program correctness, Program 6.1 requires that a lock variable be observed in a manner that guarantees that no two processors can be granted the lock concurrently. The instructions that implement LOCK may be rather costly in performance because they may need to exchange messages with some or all processors to determine when a LOCK instruction has completed. Once a processor enters a critical section, instructions can be reordered in any way that does not violate the correct execution of a sequential uniprocessor program. The instructions can be observed externally as happening in any order produced by such an execution. However, the instructions in the critical section must not be observed as if they executed before the LOCK at the beginning and after the UNLOCK at the end. In other words, whatever optimizations take place on the variables in the critical section, they must still leave the variables protected by the LOCK and UNLOCK. Likewise, optimizations may be freely applied to ordinary instructions executed outside the critical section, provided that optimizations do not have the effect of moving statements into the critical section from outside.

A number of techniques that have been proposed to achieve this behavior are in the literature. An early proposal that has influenced much of the later work is the paper by Dubois, Scheurich, and Briggs [1986], which suggested a scheme known as *weak consistency*. The basic idea has been expressed above, but more specifically the proposal is to constrain instructions as follows:

1. Special (Synchronizing) instructions are sequentially consistent among themselves. That is, there is a global ordering of all synchronizing instructions. The order is some merge of the sequences of synchronizing instructions issued by individual processors. All processors see the synchronizing instructions in this order.
2. Ordinary READs and WRITEs that execute after a synchronizing instruction in a serial program must await the completion of the synchronizing instruc-

tion before they can be initiated. They cannot appear to be executed before the synchronizing instruction.

3. Ordinary READs and WRITEs that execute before a synchronizing instruction must complete before the synchronizing instruction can begin.

The idea is that these rules provide enough timing information to assure correctness. Correctness for Program 6.1 requires that the critical section have at most one active processor. The first rule assures that the LOCK and UNLOCK can create critical sections, and the second and third rules assure that the instructions inside the critical section stay inside and those outside stay outside as observed by any processor.

The details of the implementation of this rule are rather complex because the rules as stated are vague about what it means for an instruction to "complete." On a uniprocessor, the idea is fairly well defined. But not so on a multiprocessor. Here is a list of a variety of ways an instruction can complete a WRITE, and the technical term used by Dubois, Scheurich, and Briggs to describe each type of event.

1. Processor 1 writes the new value to a store buffer between the processor and cache. The instruction is said to have *completed with respect to Processor 1* at this point. If Processor 1 immediately reads from the same address, it must retrieve the value from the store buffer.
2. The value leaves the store buffer, obtains or discovers ownership of the cache line, and enters cache. Since we assume cache coherence, the new value of the variable is the next value in the sequence of values that the variable attains. At this point the WRITE has *completed with respect to storage.*
3. The cache coherence algorithm sends messages to other processors to update or invalidate their local copies of the cache line. Processor i receives the message and changes the state of the local copy of its cache line. The WRITE is said to be *complete with respect to Processor i* at this point.
4. All processors finish updating the copies of their cache lines. The WRITE is said to be *globally complete* at this point.

For READs, the possible completion points are slightly different.

1. Processor 1 issues a READ. The request reaches cache memory and is granted access. At this point, if the item is in cache memory, no other processor can change the value to be returned by Processor 1. We say the READ is *complete with respect to all other processors.*
2. If the item is not in cache, the request is routed to the other processors and to main memory. When it reaches Processor k there will be a point in time after which Processor k will no longer be able to change the value returned to Processor 1. At this point we say the request is *complete with respect to Processor k.*

3. Eventually, no processor is able to change the value of the data returned. We say the READ is *complete with respect to all processors.*
4. At a later time, no processor that has issued a READ to the same address will be able to see an earlier value of the variable. At this time we say that the READ is *globally complete.*

Assume that a processor waits for each instruction to complete in some sense before issuing the next instruction. If that notion of completeness is complete in storage or complete with respect to the issuing processor, then programs like Program 6.2 and Program 6.3 will not be sequentially consistent as we demonstrated earlier. If the notion of completeness is global completeness, the programs will be sequentially consistent but the processor performance will be abysmal. If completeness is in respect to some processors but not all processors, the programs will not be sequentially consistent.

Weak Consistency assures the correctness of parallel programs by placing tight restrictions on synchronizing instructions and loose restrictions on ordinary instructions, with the idea of achieving good performance on ordinary instructions by means of processor optimizations when possible. The protocol is based on the following principles:

1. Each processor must assure the global completeness of a synchronizing instruction before initiating the next synchronizing instruction. This is sufficient to assure that synchronization instructions satisfy the stringent requirements of sequential consistency among themselves.
2. A synchronizing instruction that occurs in an execution sequence after ordinary READs or WRITEs must wait for the outstanding ordinary READs and WRITEs to complete with respect to all processors before it can initiate.
3. READs and WRITEs that occur in the execution sequence after a synchronizing instruction must wait until the synchronizing instruction is complete with respect to all processors before they can initiate.

The first constraint creates sequential consistency among synchronizing instructions, and the last two constraints safely keep ordinary instructions from leaking into or out of critical sections that are bounded by synchronizing instructions.

This proposal has a performance advantage over Lamport's proposal because with weak consistency, ordinary instructions can be executed at a maximum rate except in the vicinity of synchronizing instructions. Synchronizing instructions will execute at the slow rate forced by sequential consistency. So performance on balance may be good, yet the programming model enables developers to write correct parallel programs and prove their correctness. A serious issue is the implementation of the tests for global completeness, both for reasons of performance and cost.

Testing global completeness of WRITEs requires that the issuing processor receive acknowledgments from the messages sent by the processor to update or invalidate their caches. For a bus-based multiprocessor, this occurs during the WRITE bus cycle, and is a negligible performance or cost component. For other structures, this requirement contributes both performance degradation and extra cost but it is tolerable. Testing global completeness of READs is potentially more difficult. How can any processor be sure that no other processor can read an earlier value of a variable? One way to assure this is to halt all activity until all WRITEs are globally complete. At this point all changes to memory have been recorded, so that no processor can see a different value from what is recorded in memory. This implementation of global completeness is very costly and leads to severe performance degradation in programs that access shared variables frequently.

The literature that followed the paper by Dubois, Scheurich, and Briggs [1986] has sought less constraining hardware restrictions to avoid performance degradation. The proposals seek ways to put in just enough constraints to provide the tools for writing correct parallel programs and proving their correctness. One of the more appealing solutions proposed is *Release Consistency* [Gharachorloo, *et al.* 1990]. This proposal suggests using synchronizing primitives RELEASE and ACQUIRE together with ordinary instructions, and in this sense it follows the spirit of Dubois, Scheurich, and Briggs [1986]. RELEASE writes a shared variable, ACQUIRE reads a shared variable, and both operations are synchronizing operations so that special tests are used to determine when they are complete. They work together in a program to establish ordering when that ordering is necessary. When a RELEASE on Processor 1 executes before an ACQUIRE on Processor 2, then all ordinary instructions executed before the RELEASE on Processor 1 are guaranteed to be completed before any ordinary instructions are initiated on Processor 2 after the ACQUIRE. Specifically, the proposal assumes the following implementation:

1. READs and WRITEs preceding a RELEASE must be complete with respect to all processors.
2. An ACQUIRE that precedes READs and WRITEs must be complete with respect to all processors before the READs and WRITEs can initiate.
3. Any WRITEs to memory produced from synchronizing instructions issued by Processor *i* are observed by other processors in the same order in which they were generated. However, sequential consistency is not required. The WRITEs by Processor *i* and *j* may appear to be interleaved with each other in different ways by different processors.

The requirements relax the global completeness conditions of Weak Consistency in that the completeness condition for READ is simple to check. A processor that issues a READ deems it to be complete with respect to all processors when the READ returns a value, because at that point it is too late for any other processor to change that value. For WRITE operations, global completeness and completeness with respect to all processors are the same condition, and both consistency models use this condition. So for both Weak Consistency and Release Consistency, a processor that issues a WRITE must wait at a RELEASE until the result of that WRITE is visible at all other processors. This may entail waiting until an acknowledge message is received from each processor that holds a copy of the variable written.

The synchronization instructions for Weak Consistency are two-sided synchronizers in that they wait for all previous accesses to complete and prevent all subsequent accesses from starting. The RELEASE and ACQUIRE operations are one-sided. RELEASE waits for all previous accesses to complete, and ACQUIRE holds off all subsequent accesses. Neither instruction constrains both previous and subsequent accesses.

Program 6.4 is a variation of Program 6.1 to show the LOCK statement of Program 6.1 implemented as a LOCK_AND_ACQUIRE and the UNLOCK statement of Program 6.1 implemented as a RELEASE_AND_UNLOCK. This

Program 6.4 This program uses ACQUIRE and RELEASE primitives to assure program correctness of a critical section in multiprocessors that do not satisfy sequential consistency for ordinary accesses. It is similar to Program 6.1, except that the ACQUIRE and RELEASE primitives constrain the instructions within the critical section from executing outside the critical section when processors are permitted to optimize the order of execution to obtain higher performance.

```
 Procedure Add_to_Sum(var Global_Sum: Real, Shared;
   Global_Sum_Lock: Lock, Shared; Local_Table: array of Real );
 var
   i: integer;
   Local_Sum: real;
 begin
   Local_Sum := 0.0;
   for i := 1 to Max do
     Local_Sum := Local_Sum + Local_Table[i];
 {* The next statement waits until the lock can be passed, then performs an ACQUIRE. At
   most, one processor in the system can pass an unlocked control variable. The variable
   instantly locks, and prevents further access until it is unlocked. At the end of the
   critical section, the variable is unlocked immediately after a RELEASE. The RELEASE
   assures that the actions in the critical section are completed and visible to all
   processors before it itself completes. *}
   LOCK_AND_ACQUIRE(Global_Sum_Lock);
   Global_Sum := Global_Sum + Local_Sum;
   RELEASE_AND_UNLOCK(Global_Sum_Lock);
end; {* Procedure Add_to_Sum *}
```

makes explicit the synchronization of ACQUIRE and RELEASE. The LOCK statement creates a critical section. As we learned earlier, only one processor at a time can pass the lock in the critical section. The LOCK_AND_ACQUIRE statement assures timing consistency. It prevents any statements within the critical section from appearing to execute before the lock is set. Specifically, with RELEASE and ACQUIRE operations the following conditions hold:

1. On Processor 1, within a critical section the WRITE to *Global_Sum* is visible to all processors before the RELEASE_AND_UNLOCK is completed.
2. The RELEASE_AND_UNLOCK that ends the critical section on Processor 1 is visible to all processors before Processor 2 executes the LOCK_AND_ACQUIRE that enables Processor 2 to enter the critical section.
3. Within the critical section of Processor 2, all memory accesses take place after the completion of the LOCK_AND_ACQUIRE. Hence, this assures that the READ of *Global_Sum* in the critical section returns data visible at the completion of the prior LOCK_AND_ACQUIRE.

Consider what happens when two processors execute Program 6.4 in parallel. Assume that Processor 1 gains entry to the critical section first, and Processor 2 follows. The actual sequence of events must be the update of *Global_Sum* on Processor 1, the RELEASE_AND_UNLOCK on Processor 1, the LOCK_AND_ACQUIRE on Processor 2, and the update of *Global_Sum* on Processor 2. This timing imposes an ordering between the WRITE in the first critical section and the READ in the second, and guarantees that the WRITE happens before the READ.

To write correct programs using Release Consistency, Weak Consistency, or any other consistency model, the first step is to identify all places in the program where timing variations can produce inconsistent answers. Within any one processor, we assume that the processor executes programs in a way that appears to satisfy normal program order. Hence, if the program contains the instruction sequence WRITE X, READ X, the result of this execution must be that READ receives the results of the immediately preceding WRITE. The processor may actually reverse the order of execution, producing the order READ X, WRITE X at an execution stage, but interlocks in the processor must assure the execution is performed by returning to the READ the value stored by the WRITE.

The timing variations that can cause problems are those where the programmer assumes something about the relative order of READs and WRITEs issued from different processors to a shared variable. The actual sequence of accesses by those two processors must be consistent with the assumptions underlying the program. To make this clear, Gharachorloo *et al.* define *competing accesses* to be unordered accesses by two or more processors to the same location, and at least one of the accesses must be a WRITE. By "unordered" they mean that timing variations could cause the accesses to occur in any order.

Competing accesses return values that may be unexpected and incorrect, because the answers are unpredictable. A correct program must not depend on the outcome of competing accesses.

To write a correct program, one has to find all competing accesses. Presumably this is done mechanically by an analysis program. By placing RELEASEs and ACQUIREs in the program, the accesses can be ordered so that they are no longer competing. Consider an example in which a READ X on Processor 1 is followed in time by a WRITE X on Processor 2. Due to buffering, the WRITE X might change the value of X actually returned to Processor 1, even though the programmer believes that READ X occurred earlier than the WRITE X. Hence, these two instructions are competing accesses. The analysis program identifies the competing access, and the programmer removes the competition by inserting a RELEASE after the READ X and an ACQUIRE before the WRITE X. When the RELEASE occurs before the ACQUIRE, the timing of events assures that they occur in the order READ X, RELEASE, ACQUIRE, WRITE X, and the WRITE X cannot change the value reported to the READ. To assure that the RELEASE and ACQUIRE occur in the order shown, the program can use other operations in conjunction with them such as LOCK and UNLOCK. For Program 6.4, the ordering of RELEASE and ACQUIRE is done by using a RELEASE_AND_UNLOCK and LOCK_AND_ACQUIRE.

This ordering of events is sufficient to assure correct operation and to prove correctness even when sequential consistency does not hold for all instructions. Thus, programmers can reason about timing in a program by relying on RELEASE and ACQUIRE to order things between processors, and to hold off execution long enough to ensure that events truly take place before the RELEASE and after the ACQUIRE.

6.6 Summary

This chapter treats multiprocessors from a performance and topological point of view. The fundamental advantage of the multiprocessor architecture is its generality. Algorithms for such systems are much less constrained than are algorithms for vector and continuum-model computations because the individual processes in execution need not be identical or nearly identical.

The disadvantage of a multiprocessor architecture is that performance relies strongly on replication of hardware, but replication introduces serious problems regarding cost and contention. Programming complexity is greatly increased because of matters regarding synchronization and the correct use of shared data.

The negative factors tend to make multiprocessors most attractive for architectures with a small number of processors. The problem size is also important. To keep overhead low compared to useful computation, multiprocessors are best suited for large problems that cannot easily be treated on a single processor. Because of the extra complexity and overhead cost introduced to support parallel execution, multiprocessors become less attractive for dealing with problems that are solvable in reasonable time on a uniprocessor. Breakthroughs in languages and operating systems for multiprocessors could enhance the relative attractiveness of multiprocessors by eliminating the complexity that now falls on the programmer, but, to tap the potential power of the multiprocessor, the breakthroughs must necessarily provide high efficiency as well as complexity reduction.

For the near future, the likelihood of success in multiprocessor systems is assured for systems with a small number of processors. Chances for success diminish rapidly as N approaches 100 to 1000. It will take the efforts of many talented researchers pushing at the frontiers of computing research to make the 1000-processor system a cost-effective reality.

Our comments here suggest that overhead and communications costs have to be held to a minimum to achieve that reality. The hardware and software technology to keep those costs low is just developing. We expect new ideas for both multiprocessor hardware and algorithms to emerge in the next few years to help shape future architectural developments.

Exercises

6.1 Consider the performance model expressed by Eq. (6.1). Suppose the two processors have unequal speeds and that Processor 1 is α times faster than Processor 2. What is the optimum distribution of tasks to processors?

6.2 The model expressed by Eq. (6.2) is suitable for a system in which transmission time is independent of the number of processors. The cost of communication is a fixed constant C, and the formula multiplies this cost by the number of communication transactions. In a token ring, the time of transmission increases with the number of processors. Develop a model that reflects this characteristic of token rings, and find the optimum task allocation for your model.

6.3 The purpose of this exercise is to find a performance model that fits a realistic program. Consider Program 5.1 (Section 5.2). The innermost pair of loops updates a rectangular region of a matrix. The outer loop repeats this operation N times. To answer the questions that follow, ignore the cost of synchronization and count only the communications costs for data.

a) Partition the problem so that each row of the matrix lies totally within one processor. Determine the processor-to-processor communication transactions that have to occur within the algorithm. If there is no broadcast capability, how many communications occur during the algorithm? Compare this to the number of times that the innermost loop is executed on a serial computer and on the multiprocessor you are modeling.

b) If your architecture supports a one-cycle broadcast transaction in which a transmitting processor can send a common message to all listeners, how does this facility change your answer to *a*?

c) Let $N = 10$, and $R/C = 1$. What is the optimum distribution of tasks to processors for your system with a broadcast capability?

6.4 Repeat Exercise 6.3, but this time assign each column of the matrix to lie totally within one processor. Compare your answers for row and column assignments and discuss how the storage format affects the optimum way to distribute tasks among processors.

6.5 The purpose of this exercise is to investigate the effects of synchronization. For the row-oriented data structure of Exercise 6.3, reexamine Program 5.1 and discover where synchronization is required. That is, find where processors have to wait for events in other processors before they can proceed. Alter the performance model of Exercise 6.3 to account for the synchronization operations required.

6.6 Assume that the matrix of Program 5.1 is stored in N processors with one column in each processor of a multiprocessor. Let each column be updated in parallel when the subarray is updated. At the end of the update, assume that synchronization is done by means of a shared semaphore resident in Processor 0. Before an iteration begins, the variable is initialized to a value equal to the number of active processors in the forthcoming iteration. As each processor completes its work, the processor gains exclusive access to the shared variable, decrements the variable, then releases exclusive access. If a processor produces the value zero after a decrement, it initiates the next subarray update. Otherwise, processors become idle after decrementing the shared variable.

a) For $N = 16$, 32, and 128, determine the values of parameters r and h in Eq. (6.23) for a multiprocessor based on a crossbar-interconnection scheme. From these parameters, compute the maximum generation rate for memory requests.

b) Consider the question in *a* for a multiprocessor based on a bus interconnection. For this system, the point of contention is the shared bus rather than the memory system. Extend the model of *a* to cover all sources of bus contention to find a maximum rate for generating requests similar in intent to Eq. (6.24).

c) Consider the same problem executed on a machine with a shuffle-exchange network and the capability of performing Fetch-and-Add. Find the maximum rate for generating requests for this architecture for Program 5.1.

6.7 The structure of Program 5.1 requires access to both rows and columns of a matrix. Consider a very simple algorithm that accesses a matrix by two scans of the matrix. In the first scan, the matrix is accessed by rows. In the second scan, the matrix is accessed by columns. The matrix is $N \times N$.

a) For a crossbar-based multiprocessor with N processors and memories, show how to store the matrix to minimize the time for the required forms of access and state how much time is required to complete the two scans.

b) Repeat *a* for a bus-based multiprocessor.

6.8 The purpose of this question is to investigate the behavior of a multiprocessor in the absence of cache coherence.

a) Assume that a multiprocessor has caches with every processor and uses a write-through strategy for all WRITEs, but all READs first check the cache for the presence of a datum. Assume that there is no hardware support for cache coherence. Confirm that when two programs attempt to increase the same variable concurrently, it is possible to obtain incorrect results. For your program model, use Program 6.1 with critical sections removed.

b) The LOCK and UNLOCK statements in Program 6.1 create a critical section in the program. Assume that these are implemented in a way that guarantees that at most one program can enter the critical section. Now there is no failure mode due to concurrent updates of a shared variable. Show how the program fails if there is stale data in the cache.

c) Write some program instructions that eliminate a stale value of the shared variable to protect against the failure described in part *b*. Do your instructions depend on the number of sets and the associativity of the cache? Comment on the ease or difficulty of writing such instructions in a high-level language, and of the portability of the instructions from one type of processor to another.

6.9 The purpose of this exercise is to investigate the failure mode of a cache-coherence scheme that does not have write ownership.

a) Assume the presence of a write-invalidate cache-coherence mechanism that permits multiple concurrent writers. Assume that two processes execute Program 6.1 concurrently, and both processes execute the LOCK statement on the same cycle. Assume that the LOCK statement is implemented by reading the value of a lock variable, and setting that value to 1 in a single machine cycle. If the prior value of the variable is 1, the instruction repeats continually until it reaches a point at which the prior value is 0. The UNLOCK statement sets the value of the lock variable to 0 unconditionally. The lock variable can be cached.

Consider the events that take place on the underlying hardware, and consider the possible outcomes. Among the outcomes, determine if it is possible for both processes to enter the critical section and if it is possible to set the lock and have neither program enter the critical section.

b) Repeat part *a* using a write-update protocol. Does the protocol make a difference?

c) Now assume that the lock variable is noncacheable, so all accesses to the variable must go to main memory. Show that there is no failure mode if a LOCK statement can be executed in a single memory cycle in which no intervening accesses by other processes are permitted. Show that a failure mode is possible if a LOCK statement requires separate cycles for the READ and WRITE, and intervening accesses to the lock variable can occur between these.

6.10 Among the cache strategies considered in the text are write-through and write-in cache strategies that respectively write back new results to main memory immediately or hold them in cache indefinitely until they are flushed from cache. For cache coherence, two strategies studied are write-invalidate and write-update. Show that all combinations of these strategies can coexist in one system by showing that each cache line can have status bits that indicate which combination of strategies to apply.

a) Describe the status bits in a cache line that control the strategy to apply.

b) Discuss how the various combinations are implemented when a WRITE occurs and the item is in cache.

c) Discuss how the strategies affect what happens when a line is flushed from cache.

6.11 The purpose of this exercise is to consider the relative performance of write-invalidate and write-update cache coherence protocols.

The difference in performance between write-invalidate and write-update protocols depends on the number of messages sent from one processor to another to maintain cache coherence. Assume that for both protocols, a WRITE instruction generates such a message if the status bit of a cache line shows that another cache may contain a copy of the line. No message is sent if the status bit shows that the line is held exclusively by the local processor.

Messages are also sent in conjunction with ownership. In order to issue a WRITE instruction, a processor must have write ownership of a line. If it does not have ownership, the processor must request ownership and obtain the current copy of the line from the owner, which takes another processor-to-processor message.

a) Consider a multiprocessor system in which processes continually execute Program 6.1. Assume that the critical section contains 10 instructions, that the LOCK and UNLOCK statements each are one instruction, and that 1000 instructions occur between critical sections. The LOCK statement causes a processor to execute the statement repeatedly until the LOCK can be passed. Both LOCK and UNLOCK require write ownership. Model the behavior of two processors and determine the average number of processor-to-processor messages per instruction for both protocols. Note that both processors can conflict at a critical section, but that after the first conflict, no more conflicts occur. For your analysis, assume that no conflicts occur at critical sections.

b) Extend your model to analyze the number of messages generated as the number of processors increase. For what value of *N* are there enough processors to assure that two or more processors conflict at a critical section? What is the difference in the behavior of the two protocols when multiple processors conflict at the critical section?

c) Modify the implementation of LOCK so that it is two instructions long. The first instruction is the conditional branch that reads the lock variable repeatedly until the value of the variable is 0. The second instruction is the lock instruction used in part *a* in which the lock variable is read, tested, and rewritten in a single cycle. What is the difference in the behavior of the protocols when two or more processors conflict at the critical section?

6.12 The purpose of this exercise is to examine the difference in the consistency models.

Consider Program 6.1 when executed on a computer whose interconnections are based on the shuffle-exchange network or crossbar network, neither of which assure sequential consistency. In Program 6.1 the LOCK, UNLOCK, and update of Global_Sum are assumed to be the only statements that access global variables.

a) In order for the execution of the program to be sequentially consistent, each access to a shared variable has to be serialized according to some global ordering. Discuss how to achieve a global ordering of all accesses on the computer so that all processors observe all READs and WRITEs in the same order. You should incorporate extra hardware into your system to implement sequential consistency. Assume that sequential consistency must be maintained on all accesses to shared variables, but that it is not necessary to maintain sequential consistency on local variables such as *Local__Sum* in Program 6.1. Your scheme should have a zero or small performance penalty for accesses to local variables.

b) Now consider release consistency instead of sequential consistency. Discuss how to implement release consistency on your system so that Program 6.4 operates correctly.

c) Make qualitative comparisons of the cost of implementation of sequential consistency and of release consistency, and compare the performance penalties of your implementations of Programs 6.1 and 6.4.

7

Who depends upon another man's table often dines late.

—John Ray, 1678

Multiprocessor Algorithms

This chapter explores the means for programming multiprocessors for high performance. A major portion of the chapter is dedicated to efficient mechanisms for ensuring the correct execution of programs. Our approach is to look at the easy parallelism first. The obvious ways to execute in parallel produce the bulk of the gains for most applications.

When one attempts to wrest the ultimate performance from a parallel process, it becomes necessary to explore more sophisticated notions. This chapter shows that search algorithms, for example, yield rather poor speedup when the programmer naively assigns dependent tasks to different processors. This is the case, for example, if a search terminates when any processor finds a solution, and the search space is divided among all processors.

We show a different approach that uses parallelism rather efficiently to solve a classic optimization problem, the Traveling-Salesman Problem, in a time that on the average grows less than quadratically in the size of the problem. This may appear to be rather astounding, since the Traveling-Salesman Problem is one of the so-called *hard* (NP-complete) problems, and therefore there exists no known algorithm that solves this problem in a time that grows less than exponentially in the problem size. But theory covers the worst case and says nothing about the average case. We cover the average case for a random problem in this text, and that has a very low complexity.

Correctness of parallel algorithms requires some mechanism for handling the updates of shared variables. We introduce the performance notion of SYPS (SYnchronizations Per Second, pronounced "sips"), which is normally measured in MSYPS (MegaSYPS).

In this chapter we show how the MSYPS capacity of an architecture affects throughput. Throughput is limited both by its MIPS and MSYPS capacity and cannot exceed the throughput permitted by the more constraining of the two measures. Thus a high-MIPS, low-MSYPS machine may be outstanding at numerical operations, but can run rather poorly for applications that require a high volume of synchronizations. The MIPS measure alone suggests a high throughput, but the architectural constraint on MSYPS can prevent the potential MIPS from being realized.

7.1 Easy Parallelism

Parallelism is best used for programs that require a significant number of cycles. We have accomplished something worthwhile when we reduce a ten-day execution to one day, whereas the reduction of a ten-minute program to one minute is an equal but far less interesting speedup. We argue here that long programs almost surely contain some region of code that accounts for the bulk of the execution by being executed repeatedly for a massive number of times.

At a clock rate of 100 ns, there are on the order of 10^{12} clock ticks in a day. Consider any program that takes a full day to execute and examine where it spends the bulk of its time. If there is some subroutine or code sequence that is repeated a large number of times, say a million times, then our thesis is justified. The alternative is that no program instruction is executed more than a few times. Consider such a program.

At ten ticks per instruction and as many as ten repetitions of an instruction, we find that the program must contain about 10^{10} distinct instructions to execute for one full day. Such a program would indeed be unusual because of its gigantic size, and the effort to construct such a program would take thousands of man-years at current rates of software productivity. The program is more likely to have only 10^4 to 10^6 instructions, therefore requiring an average repetition factor of roughly 10^5 to 10^7.

With some body of instructions being repeated a million times or more, we have an opportunity for parallelism if we can spread those million executions in some way across N processors. This is a simple recipe to achieve parallelism:

1. Analyze the program for a loop or recursion structure;

2. Find the instructions that account for the most time, usually the regions repeated the greatest number of iterations;
3. Split the instruction execution of these regions across N processors, if this can be done correctly; and
4. Add synchronization and data-transmission statements as required to create a correct parallel implementation.

As an example of the application of this idea, consider Program 7.1, which revisits the Poisson calculation introduced in Chapter 4. Recall from our earlier discussions that the near-neighbor iteration is usually not the most efficient way to solve the Poisson problem. Nevertheless, iteration is what appears in Program 7.1.

Suppose, also, that we know in advance that $10M$ cycles are required for the iteration to converge. Program 7.1 shows three nested loops. The outer loop repeats $10M$ times to obtain the necessary convergence. (The fixed number of outer iterations is just a convenience for this example. Most implementations repeat the outer iteration until some convergence test is satisfied.)

In the two inner loops, each point $P[i, j]$ in a square region is updated once. The innermost loop updates a line in the region, and the next level of iteration treats the collection of lines that cover a rectangle. The outermost iteration forces the rectangle to be updated $10M$ times.

A purely sequential program updates all points in the rectangle one time before any point in the rectangle is updated a second time. To enforce this

Program 7.1 Poisson solver, serial version.

```
for k := 1 to 10 × M do
  begin
    for i := 1 to M do
      begin
        for j := 1 to M do
          begin
            P[i,j] :=
              (P[i,j+1] + P[i,j-1] + P[i+1,j] + P[i-1,j])/4;
          end; {* j loop *}
      end; {* i loop *}
  end; {* k loop *}
```

Notes:

1. Boundary conditions are held in Rows 0 and $M+1$ and Columns 0 and $M+1$ of array P.

behavior in a parallel program, we seek a scheme that uses parallel processors as effectively as possible for a single update of the rectangular region, and we perform 10*M* executions of the parallel update, with the updates occurring one after another, without any overlap among them. Figure 7.1 shows a possible execution diagram, with the number of processors busy as a function of time and the outer iteration that they are performing at any given time.

7.1.1 The do par and do seq Constructions

From a programming point of view, we need the concept of parallel and serial embedded in a language to distinguish between iterations that can be done in parallel across many processors and those that have to be done one after another. A simple way to extend a Pascal- or FORTRAN-like language is to introduce these forms of the **do** construction:

- **do par** to execute loop iterations in parallel; and
- **do seq** to execute loop iterations sequentially.

Then the form

```
for i = 1 to M do seq
  begin
    Iteration A
  end; {* do seq *}
```

produces *M* serial executions of Iteration A, whereas

```
for i = 1 to M do par
  begin
    Iteration A
  end; {* do par *}
```

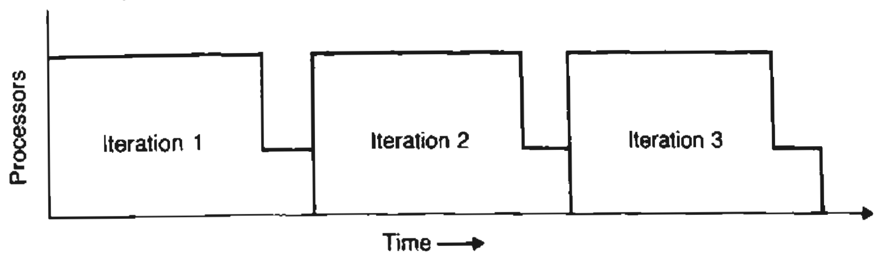

Fig. 7.1 Processors busy as a function of time. All available processors are busy until most of the work for an iteration is done. As an iteration nears completion, some processors become idle and must wait until a new iteration starts before they can resume computation.

causes all M copies of Iteration A to be alive concurrently, and any or all those copies can be executed concurrently, depending on scheduling policies and the resources available. The **do par** construction creates a separate instance of the loop body for each value of i in the range of **do par**.

To describe our findings regarding the parallel and sequential behavior of Program 7.1, consider Program 7.2, in which the two inner loops use the **do par** construction, and the outer loop uses the **do seq** construction. During the course of execution, this program creates M^2 copies of the inner iteration, one for each (i, j) pair, parcels these out among the processors, then awaits their completion. When they have completed, the program performs the same process again and continues repeating it until it is done $10M$ times.

7.1.2 Barrier Synchronization

Notice the synchronization that is implied by the **do seq** construction in Program 7.2. A processor ready to begin a new outer iteration has to be informed when all work for the last outer iteration has been completed.

In essence, the **do seq** construction has placed a barrier after each of its iterations. As many processors as can be used effectively can be allocated to a single iteration of a **do seq**, but those processors must stop at a barrier at the end of the iteration. No processor can cross this barrier until all processors performing the loop iteration have reached the barrier.

In Program 7.2, we can have as many as M^2 processors executing within a single iteration of the outer loop, and these processors have to stop and wait at

Program 7.2 Poisson solver, parallel version.

```
for k := 1 to 10 × M do seq
  begin
    for i := 1 to M do par
      begin
        for j := 1 to M do par
          begin
            P[i,j] := (P[i,j+1] + P[i,j−1] + P[i+1,j] + P[i−1,j])/4;
          end; {* j loop *}
      end; {* i loop *}
  end; {* k loop *}
```

Notes:

1. Boundary conditions are held in Rows 0 and $M+1$ and Columns 0 and $M+1$ of array P.

the implicit barrier for all to finish before any one processor can start a new iteration. We call this type of synchronization *barrier* synchronization. Although it is not used explicitly in Program 7.2, it is implicitly used at the end of each iteration of the **do seq**.

An explicit form of the barrier can be used as shown in Program 7.3 within the body of a **do par** construction. In this case, the body of the loop has three parts, Steps A through C. The **do par** creates *M* instances of the loop body, one for each value of *i*, and parcels these tasks to as many processors as are available.

In the absence of barriers, for any single iteration we are guaranteed to execute Step A(*i*), then B(*i*), then C(*i*), in that order. The order in which the steps are performed across iterations is rather arbitrary, and anything could happen. For example, we could see the completion sequence A(1), A(2), B(2), C(2), B(1), C(1). We could not see a sequence in which B(1) completed after C(1) because a loop body for a specific iteration has to be executed serially.

Program 7.3 has a barrier inserted after Step B. The effect of the barrier is to force all iterations to complete Steps A and B before any iteration continues to Step C. With the barrier in place, the sequence A(1), A(2), B(2), C(2), B(1), C(1) cannot occur because C(2) completes (and hence must have been started) before B(1) has been completed. The barrier should be inserted if Step C of each iteration depends on Steps A and B of prior iterations.

The barrier is a rather strong means for synchronizing, and it may be more severe than is actually necessary. It may be possible to use more focused methods of synchronization that can start Step C in various iterations at much earlier times. Such methods necessarily have the ability to sense when specific conditions are satisfied so that Step C can start, which is more flexible than sensing the single condition that all processors have reached a barrier.

Program 7.3 Barrier example.

```
for i := 1 to M do par
  begin
    Step A(i)
    Step B(i)
    Barrier;
    Step C(i)
  end; {* i loop *}
```

Notes:

1. The Barrier forces all iterations of *A* and *B* to complete before any iteration of C is started.

7.1.3 Performance Considerations

Given these basic notions of parallel and sequential execution of loops, let us examine the performance aspects of the parallel code. For the moment, let us ignore the specific details of initiating a parallel task at the beginning of a **do par** and of handling a barrier, if any, associated with the **do par**. Our objective is to determine the *R/C* ratio for a program so that we can relate the results of Chapter 6 to multiprocessor algorithm development.

In Program 7.2, a single task corresponds to the one statement of the innermost loop. This statement takes roughly six instructions, consisting of a LOAD, three ADDS, a SHIFT or DIVIDE, and a STORE. Address calculations might be required as well, but they might be avoidable if the address computations required can be done totally by means of the effective-address mechanism without requiring additional instructions. We also should include some additional time to charge to the iteration for the calculation of the values of i and j to use for this particular iteration. In total, roughly ten instructions are necessary to perform the iteration. This corresponds to R, the run time.

The overhead and communication encompassed by C includes the work required to generate the task, to enqueue it while waiting for a processor, to dequeue it when a processor becomes available, and to log the completion of the task so that some barrier can be passed when all tasks are completed.

We may be fortunate enough to avoid an ENQUEUE/DEQUEUE pair, but there have to be some instructions to generate and terminate the task. A very low estimate for this overhead is two instructions for each of generation and termination. A more realistic estimate is hundreds, possibly thousands, of instructions.

The ratio *R/C* might be as high as 2 or 3, and it could be as low as 1/100 or 1/1000. For most of the models mentioned in Chapter 6, these ratios do not support a good deal of parallelism. Depending on the architecture and the ratio, the fastest implementation of Program 7.2 uses only one processor or possibly just a few processors. But this is still rather optimistic because our earlier models ignore the effects of synchronization. Synchronization produces further degradation that biases the best solution towards fewer processors.

To be more specific, consider how synchronization affects a single task in Program 7.2. The task has to be generated, enqueued, dequeued, and terminated. The enqueue, dequeue, and terminate processes are likely to involve shared variables that have to be updated. The task-generation process might introduce its own overhead as well if it, too, updates shared variables.

Let us count the updating of a shared variable as a basic operation that we call a *synch*. Then one task of Program 7.2 requires three synchs (for enqueue/dequeue/terminate), plus roughly ten instructions for task generate, loop body, and task terminate. Most multiprocessor architectures are highly constrained in how synchs are implemented, and the number of synchs that can be performed in parallel is typically rather limited, sometimes as few as one. An exception to this is an architecture with the combining switch described in Chapter 6, such as the IBM RP3 and NYU Ultracomputer architectures.

To understand the synch problem more thoroughly, consider a bus-oriented multiprocessor that uses a READ/MODIFY/WRITE operation on the bus to perform a synch. Then at most one synch per cycle is possible. For a cycle time of 100 ns, this limits performance to at most 10^7 SYPS (synchs per second), or 10 MSYPS.

If in one cycle the multiprocessor can execute one instruction in each of N processors, then the performance of the composite system is 10N MIPS for instructions, but only 10 MSYPS for synchs. The MIPS rate is N times greater than the MSYPS rate. Our example program demands roughly two or three instructions per synch, so that for N greater than 3, the system becomes saturated at the synchronization interface; otherwise, the system is saturated at the instruction-execution interface.

A combining switch provides a mechanism for supporting synchs in parallel, and thereby it provides an MSYPS rate more nearly on the order of 10N MSYPS for a system with a 100 ns clock. The coefficient need not be 10; it may be considerably less. The point is that the sustainable MSYPS rate grows with N, and it thereby provides a means for breaking the synch bottleneck.

Architectures that do not have a combining switch or an equivalent mechanism for executing synchronizations in parallel are subject to a saturation phenomenon depicted in Fig. 7.2. The assumption in this figure is that there is a fixed maximum MSYPS supportable by the system, independent of the number of processors. As processors are added, the MIPS rate of the system grows linearly with the number of processors, but the MSYPS rate is fixed. Eventually the MSYPS demand reaches the limit, and no additional speedup is possible as new processors are added.

The figure shows linearly increasing speedup until ten processors are in the system; thereafter speedup remains at the saturation limit of ten as new processors are added. Two curves are shown—an idealized piece-wise linear curve that reflects the bounds on speedup, and a curve that falls below this bound, which suggests what might be observed in actual situations. The true curve shows speedup falling off with additional processors because overhead tends to increase and MSYPS capacity remains at the fixed limit as new processors are added.

We have reached an interesting challenge for a computer architect. Suppose that an application such as Program 7.2 is implemented for a multiprocessor, and performance turns out to be sharply restricted because of an MSYPS bottleneck. What avenues are open to the architect to improve performance? Here are three obvious directions to follow:

1. Increase R/C and thereby do more computation per synch.

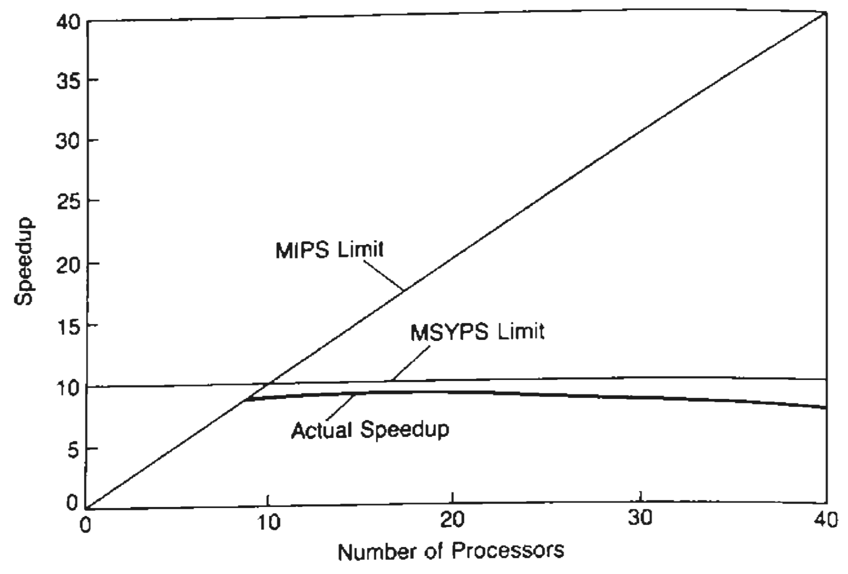

Fig. 7.2 Speedup curves.

2. Balance the system by making architectural or program changes to increase the MSYPS rate of the architecture.
3. Balance the system by reducing the MIPS rate of the processors.

The first approach is the easiest and most cost-effective. We can substantially improve performance for essentially no cost in hardware or software by increasing granularity. This is the preferred solution that is discussed at some length in this section.

As an example of the second approach, the architect can build into the architecture mechanisms that support a high MSYPS rate. The combining switch is an approach in which the MSYPS rate increases linearly with the number of processors, but other techniques that may raise the MSYPS rate high enough for specific applications are also possible. For example, the architect can incorporate a high-speed specialized processor for synchronizations that does nothing but manage locks and the updating of shared data. In a multiprocessor, the architect might also include a hardware scheduler/dispatcher for task and processor management.

The third approach, reducing the MIPS rate of the processors, corrects system imbalance but reduces overall throughput. The idea here is that if system imbalance results in idle processors, one may be able to obtain nearly equal speed by using less expensive slower processors. This approach attempts to exploit the cost disparity between low-speed and high-speed technology, and can be successful if the change in throughput by reducing the speed or number of processors is sufficiently low compared to the reduction in the cost of the system. The idea is to change from an inefficient system to a much less expensive system of slightly lower capacity by exploiting higher efficiency.

7.1.4 Increasing Granularity

To continue this discussion, let us see how easy it is to increase *R/C* for Program 7.2. The granularity assumed in the program is that there is one assignment statement per task. To increase granularity we can group several statements together, as suggested by Program 7.4.

Program 7.4 is identical to Program 7.2 except that the innermost loop contains the phrase **chunksize** 50. This phrase instructs the compiler and operating system to group 50 successive index values into each task, instead of assigning one index value to each task. The last task to be assigned receives whatever index values remain, which may be fewer than 50. With the chunk-size set to 50, *R/C* is 50 times greater for Program 7.4 than for Program 7.2, and the MSYPS requirement is reduced by a factor of 50. Of course, the parallelism available is also reduced by a factor of 50, but the point is that the reduction in parallelism might be quite tolerable if it were not usable in the first place.

Program 7.4 Poisson solver, parallel version with chunking.

```
for k := 1 to 10 × M do seq
  begin
    for i := 1 to M do par
      begin
        for j := 1 to M do par chunksize 50;
          begin
            P[i,j] := (P[i,j+1] + P[i,j-1] + P[i+1,j] + P[i-1,j])/4;
          end; {* j loop *}
      end; {* i loop *}
  end; {* k loop *}
```

Notes:

1. Boundary conditions are held in Rows 0 and $M+1$ and Columns 0 and $M+1$ of array *P*.
2. The phrase **chunksize** 50 forces iterations to be parceled out to processors in chunks of size 50, with each of the iterations in a chunk performed sequentially. Different chunks can be executed concurrently on different processors.

For example, consider the potential for parallelism when M in Programs 7.2 and 7.4 is equal to 100. The two inner loops create 10,000 tasks in Program 7.2. The number of tasks actually created depends on the program, not on the architecture. If the architecture has fewer than 10,000 processors available, as is likely to be the case, then the excess tasks created will probably be enqueued and dequeued or generated on demand, but in any case will result in 10,000 instances of overhead related to their management. Program 7.4 gives the programmer the ability to reduce the overhead by controlling how many independent tasks are created, as well as the *R/C* ratio for those tasks.

For the example we are considering, Program 7.4 creates 200 tasks, which is appropriate for architectures with 200 or more processors. If the architecture has fewer than 200 processors, the chunksize should be made even larger, and it is realistic for the chunksize to be computable dynamically to be a function of the number of the processors actually available for execution of the loop body.

The purpose of a small granularity, after all, is to increase the available parallelism, but there is no point to increasing parallelism beyond the amount that can be exploited. Granularity should be set no smaller than the size that creates enough tasks to fill available processors, and perhaps even this size is too small if *R/C* for that granularity is below the break-even point for the processors available. The point in making the chunksize selectable by the programmer is that the programmer can experiment with grain size to find some optimum size for a given application and architecture.

Granularity is only one of several factors that the programmer has to consider. We have not addressed the issues regarding local and global storage and allocation of data to reduce memory contention. When the programmer chooses a granularity by choosing a chunksize, the programmer is actually binding together various iterations and is thereby creating an environment in which some data can possibly be reused several times in a local context before being returned to a global memory. In this environment, the task can be structured as follows:

1. Acquire locks as required for global variables to be updated.
2. Read variables from global memory to local memory.
3. Perform the computation, updating the local variables.
4. Update the global variables from the local copies of the variables.
5. Release the locks on the global variables.

While the computation is being executed, contention with other processes is held to a minimum because all accesses are to local memory. However, locking and the synchronization overhead required to obtain and release locks can degrade performance. Much depends on the likelihood that processors will be left idle while waiting for locks to be released.

In creating a large task by choosing a large chunksize, the programmer actually has more flexibility than is shown in Program 7.4. That program provides only for creating tasks by grouping together iterations that fall on a single row of the square array. The program can be reorganized so that chunks fall instead along columns, or in rectangular or square subarrays.

The structure that forms the best possible chunk has a good granularity and can operate on the data for that chunk with minimum interference with processors that operate on their chunks. The amount of interference expected to occur depends on the architecture and the allocation of data to memory modules within that architecture. The designer of the architecture has to be aware of the control choices available to the programmer and should create an architecture in which one or more of those choices leads to efficient execution across a range of important problems.

The programmer has a rather powerful means for controlling the size of *R/C* by controlling chunksize and by selecting which statements are grouped together within one chunk. If the chunksize is fixed for an architecture, as several proposals for fine-grained architectures have suggested, the programmer loses the flexibility to adjust the *R/C* ratio to obtain maximum performance. First- and second-generation multiprocessors should leave the ratio in the hands of programmers until sufficient experience is obtained to build machines with optimal or near-optimal *R/C* ratios.

The second technique for eliminating an MSYPS bottleneck is to reduce the cost of a synchronization, or equivalently, to increase the MSYPS rate of the architecture. This subject is sufficiently complex to warrant its own section within this chapter. We defer discussion at this point and explore the subject in depth later.

The last technique achieves balance within a system by slowing down the processors relative to the synchronization mechanism. Thus, the MIPS rate of the system is reduced while the MSYPS rate is fixed, and this yields a better balance if MSYPS are not well matched to the initial value of MIPS.

Figure 7.3 shows speedup as a function of the clock period as the clock is slowed. Note how speedup in this system increases as the processors become slower. Recall that speedup is a measure of the speed of an *N*-processor system as compared to a system that has one processor identical to any one of the *N* processors. Figure 7.3 is plotted for $N = 100$. Since clock period increases along the x-axis, the processors at the right-hand side of the figure are slower than the processors at the left-hand side of the figure.

The figure shows that the speedup obtained from 100 processors is greater for slow processors than for fast processors. However, speedup is not the same as performance. The performance from 100 fast processors is greater than the performance available from 100 slow processors, even though speedup is less for the fast processors. On the left side of the diagram, the fast processors are not well matched to the slow synchronization mechanism, and many are left

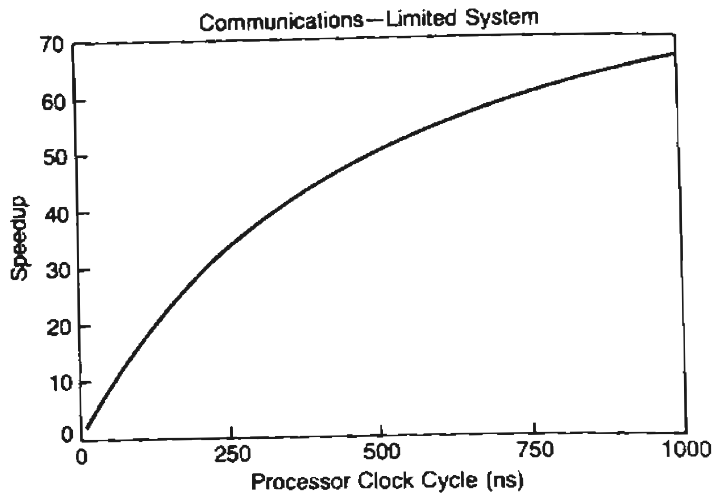

Fig. 7.3 Speedup versus clock period.

idle during a computation. Adding new processors to this system does not improve performance very much, so speedup is relatively low.

As we move from the left to the right of the figure, the bottleneck in the system shifts from the synchronization mechanism to the processors themselves. When the processor performance is the chief component of the bottleneck, then by adding new processors, the bottleneck is reduced so that speedup tends to increase. Cvetanovic [1985, 1987] made this observation in regard to her study of an RP3-like architecture, but the phenomenon holds in general for systems that have two or more potential bottlenecks.

The lesson to be learned from Fig. 7.3 is that the architect should select a design point in which bottleneck capacities are close to being in balance. For the multiprocessor architecture, the maximum system MIPS and MSYPS rates should be balanced with respect to each other to match the demands of most workloads. If the system is out of balance by being on the left side of Fig. 7.3, the processors are too expensive for the system performance they give. On the right side of the figure, the processors themselves are the bottleneck, and additional speed can be obtained by faster processors.

7.1.5 Initiating Tasks

One topic of importance that we have overlooked thus far concerns the mechanism for initiating individual tasks. If, in Program 7.2 or Program 7.4, the **do**

par construction is implemented by generating the tasks one by one, then the task generation is a serial overhead that must be added to C in the R/C ratio.

Program 7.2 depicts a situation in which the inner loop requires O(N) instructions just to generate the tasks if task generation is done sequentially. Yet the tasks themselves take only ten or so instructions that are supposedly done in parallel.

This situation becomes rather comical if you observe a processor executing the **do par** and spinning off 100 tasks by executing 1000 instructions. After spending all of this time generating the work, within ten more instructions all the work is done. We have simply shifted execution time from doing the main iteration to the overhead in starting up the processors. Obviously, the R/C ratio is far too low to be useful, but more fundamental is the fact that we cannot afford to use sequential execution to spin off the tasks to be executed concurrently.

A good approach is to produce the tasks during compilation, provided that the value of N is known during compilation. Then the tasks are created once for all executions of the program. Presumably, once the tasks are created, they can be loaded in parallel into all processors, and thereby we avoid the serial time for their initiation.

An alternative approach that has somewhat higher overhead is to generate the tasks dynamically in O(log N) time by means of a binary task-generation tree. To generate the tasks for the innermost **do par** loop of Program 7.1, the root node of the generation tree generates two subtasks. The first is responsible for generating the first half of the tasks, and the second is responsible for generating the second half of the tasks. These in turn split into four subtasks, each responsible for generating a quarter of the tasks. After O(log N) steps, no additional subtasks are generated, and the tasks themselves can be generated.

The tree-generation scheme or an equivalent is absolutely essential for dynamic task-initiation. Any O(N) process for task generation can create sufficient overhead to severely impair multiprocessor performance.

The task-generation scheme appears to be an obvious requirement. Yet it has been overlooked repeatedly in the literature in serious proposals for multiprocessors. Halstead [1985] describes an interesting multiprocessor architecture called *Concert*, in which the user has explicit control of task generation. This paper describes an example of parallel sorting using the well-known quicksort algorithm, which has an average complexity of M log M for sorting M items.

The initial phase of the Halstead algorithm is a linear pass over the M items. This phase generates a collection of tasks that can be executed in parallel. Subsequent phases of the algorithm exploit parallelism rather well, but the first phase does the damage. No matter how many processors are used, the algorithm cannot run faster than O(M), thereby dooming speedup to O(log M). Halstead reports near linear speedup for a small number of processors, but as the number of processors grows close to log M, speedup must level off.

The limitation on speedup in this case is not the fault of the architecture because Concert, like many multiprocessors, supports task-generation trees. The fault lies in the data representation of the problem. The data to be sorted in this problem are presented to the algorithm as a LISP one-way linked list. The only way to inspect the data is to follow the chain of pointers from one item to the next, taking $O(M)$ time to do so.

Here is a situation in which the data representation from a serial programming language is strongly incompatible with high-performance parallel processing. Although Halstead's article articulates the strengths of the Concert architecture, it does not specifically address the weaknesses of a linked-list structure in the context of the algorithm. The data representation in this case imposes an inherent inefficiency on what otherwise appears to be an interesting and effective technique for exploiting parallelism in a multiprocessor.

The key to architectural evaluation is identifying how performance changes as a function of critical parameters such as the number of processors, R/C, and the choice of data structure. We have shown how a few simple notions provide extremely powerful tools for identifying major bottlenecks that are otherwise hidden from view.

In closing our discussion of easy parallelism, note how the example for this discussion shows the advantages of the multiprocessor over a near-neighbor SIMD machine and other various forms of vector machines.

Program 7.2 is ideal for a near-neighbor or a vector machine, as stated, but real applications are seldom as simple as Program 7.2. The boundary calculations are often rather complex, and in the more usual case, the region is irregularly shaped or has internal cavities or other structures that alter the simplicity of the solution.

Each different type of point within the region of computation requires a slightly different program. A purely SIMD machine cannot easily deal with such differences and still retain high efficiency. Each different type of point, in the worst case, requires its own program execution, done with all other processing turned off. Thus, an SIMD machine may have to perform successive computations for the points of Region A, Region B, and so on, and thereby reduce the effective parallelism available in the architecture.

The multiprocessor can produce different programs for each region and perform the computations for all regions concurrently, thus achieving greater parallelism than an SIMD architecture can achieve. We presume that the number of different programs required is a small number, such as 10 to 20, and that the execution time per iteration is equal to the longest time logged by any of the different programs. If there are k different programs to execute, then the gain of the multiprocessor over the SIMD architecture is at most a factor of k.

7.2 Synchronization Techniques

Synchronization is probably the most difficult and error-prone type of programming that exists. Its difficulty arises because it involves the understanding of the potential simultaneous actions of multiple processors. The huge number of possibilities to consider is beyond the capability of most people. Moreover, synchronization also depends on the nature of the interfaces among the multiprocessors. Many schemes have fallen because the programmers have made false assumptions about how the hardware works.

As an example of this problem, consider the landmark work of Dijkstra [1965]. At issue at that time was whether or not processors could be synchronized with just the standard operators of an ordinary programming language such as ALGOL 60. Dijkstra's solution was the first to show that this is possible for a reasonable set of assumptions. He states that this is the most difficult program he has ever written.

The statements in his program made no use of instructions that can perform uninterruptible READ/MODIFY/WRITE operations because ALGOL did not supply this operation in any form as a primitive operation. But the program did assume that the multiprocessors satisfied sequential consistency as described in Section 6.5.

As we learned in Section 6.5, there is a great performance advantage in giving up sequential consistency in multiprocessors, so that the Dijkstra synchronization algorithm fails as will any algorithm that relies on sequential consistency if the underlying system does not support the principle.

Dijkstra's synchronization solution is not important today because almost all synchronization is done with READ/MODIFY/WRITE operations of some form. These are the synchronizing instructions that make program correctness possible on such systems. In the remainder of this section we treat a sequence of five general methods for synchronizing processes. The progression moves from the least powerful to the most powerful, and the discussion suggests how the additional power can be used to obtain enhanced capabilities. The five methods treated here are

1. *Test-and-Set*: operate on a single bit.
2. *Increment, Decrement*: produce sums and differences.
3. *Compare-and-Swap*: reduce a complex critical section to a single instruction.
4. *Reservations for READ/MODIFY/WRITE*: let some types of instructions intervene during a READ/MODIFY/WRITE sequence.
5. *Fetch-and-Add*: eliminate critical sections in some cases.

The remainder of this section treats each of the alternatives in order.

7.2.1 Synchronization with Test-and-Set

The first synchronizing method uses an instruction called *Test-and-Set*, which performs the following operation:

```
Definition: Test-and-set(address, bit_ position);
  begin
    Temp := Memory[address] .bit_position;
    Memory[address] .bit_position := 1;
    Condition_code := Temp.bit_position;
  end; {* definition *}
```

The Test-and-Set instruction sets a designated bit of a shared datum to 1, and returns in the condition code the value of that bit prior to setting it to 1. The two parameters of the instruction are the address of the shared datum and the bit-position of the datum at the address that is to be tested. The notation "A.b" denotes bit position *b* of datum *A*.

This instruction has the classic form of READ/MODIFY/WRITE, which is a key characteristic of synchronizing instructions. To ensure that it can be used successfully for synchronizing, the Test-and-Set instruction must be uninterruptible. That is, once it is initiated and the READ access is completed, no other access can be made to the operand until the operand is rewritten during the second step of the Test-and-Set. If an intervening access were permitted, synchronization could fail.

Multiprocessors that have cache memories must treat Test-and-Set as a special type of instruction. Since Test-and-Set is used to update shared data, shared data held in cache must be kept consistent across all caches and with respect to main memory.

One possibility is to force accesses produced by Test-and-Set to go to shared memory and avoid the cache altogether. The companion operation that resets bits of shared operands should be implemented in a similar fashion. Another alternative is to permit shared data to be cached and to build the necessary synchronization behavior into the cache-consistency protocol.

One possibility here is to use an ownership bit in the cache directory to indicate which copy of a shared datum resident in one or more caches is the principal copy. When the READ of the READ/MODIFY/WRITE is performed, the cache that owns the shared datum passes its current value to the requestor. All processors except the requestor mark the datum as absent. When the datum is rewritten, it can be rewritten to the local cache, with the datum tag showing the datum being owned exclusively by the local processor.

Now consider how one might use a Test-and-Set instruction to implement an elementary update of a shared variable. The skeleton for a program is:

```
Lock(shared_datum);
Update(shared_datum);
Unlock(shared_datum);
```

With each shared datum or data structure, we can associate a single bit, called its *semaphore*. The Lock and Unlock statements operate on the semaphore of a datum or data structure rather than on the content of the datum. The semaphore is the traffic director that tells a process whether or not to proceed past the LOCK statement. The semaphore permits at most one process at a time to execute the code in the update region of the program. If Process *A* executes the Lock statement successfully, then all other processes must be halted there until Process *A* executes the UNLOCK statement.

The LOCK statement can be implemented in part with a Test-and-Set instruction. The Test-and-Set forces the semaphore for the shared datum to be set, whether or not it has been set before the Test-and-Set. To pass the lock, the process must see 0 returned in the condition code as the value of the semaphore just prior to the Test-and-Set.

If several processes execute a LOCK on a semaphore concurrently, the requests will be serialized and executed one by one because of the characteristics of the READ/MODIFY/WRITE operations that force this serial behavior. Given serial execution of LOCK, no more than one process of a set of concurrent requestors can observe a zero value of the semaphore and thereby move past the LOCK to the update. When one process passes the LOCK and reaches the UNLOCK, the semaphore can be returned to a 0 state and thereby permit another process to pass the LOCK statement and update a shared variable.

In terms of MSYPS, the LOCK/UNLOCK pair take at least one instruction each. The update code protected by the LOCK/UNLOCK requires two or three instructions and could be 10 to 100 instructions, depending on the nature of the update. This puts anywhere from 5 to 100 or more instructions in the serial section.

The number of serial sections executed sequentially in one second gives the MSYPS rate, which is therefore anywhere from 5 to 100 times slower than the MIPS rate of the processor. The MIPS rate is likely to be the bottleneck if its MSYPS rate is very high, for example, 10 percent or more of the MIPS rate. The bottleneck shifts to the MSYPS rate if MSYPS is relatively low, for example, 1 percent or less of the MIPS rate, depending on the application.

In multiprocessor systems the peak MIPS rate increases proportionally with the number of processors assigned to a problem, but the MSYPS rate in most architectures is a fixed limit for a system regardless of how many processors are actually assigned to a program.

If we focus on the MIPS rate exclusively and ignore the MSYPS limit, we tend to believe that by assigning more processors to a program, we are making available more machine capacity. But this is not strictly true.

Indeed, as more processors are assigned, a program has more MIPS and more memory available, but MSYPS may not be increasing at all. If this is the bottleneck, then additional processors will not result in faster computation. In fact, because of contention among processors, the LOCK/UNLOCK and the update operations on shared data tend to take longer with more processors active, with the result that computation time may increase instead of decrease as more processors are assigned to a computation.

The MSYPS bottleneck is only one of several potential sources of performance degradation. For example, consider what happens when a processor is blocked by a LOCK operation. Perhaps it can be put to use doing other useful work and continue to expend MIPS fruitfully in spite of the MSYPS bottleneck. The Test-and-Set is only half of a lock. The other half is the action taken depending on whether the lock has been granted or not. If the Test-and-Set observes a prior semaphore value of 0, then the lock has been granted, and the processor continues on to the update section. If not, there are at least two different actions that can be taken:

1. *Spin lock*: branch backward and reexecute LOCK, repeating the process until the lock is granted.
2. *Enqueue a task*: suspend the blocked process, and enqueue its status on a queue associated with the semaphore. Reassign the processor to other work currently enqueued and ready for execution.

Neither of these alternatives is particularly attractive. The spin lock wastes computer cycles and causes memory contention at the semaphore. When many processors are waiting at a semaphore, the contention causes additional cycles of delay while a process is attempting to release a lock. This tends to decrease the sustainable MSYPS rate and magnifies the effect of the bottleneck at the semaphore.

Task enqueueing appears to be efficient because it devotes available cycles to useful work. However, the overhead for ENQUEUE/DEQUEUE tends to be very high, which may well be greater in cost than the cost of the cycles lost in a spin lock. Worse yet, to enqueue a task, a processor has to access and update a shared queue pointer. This access itself involves a LOCK/UNLOCK of some kind.

If this lock is not granted, we have come full cycle and face the problem of enqueueing a task at one queue to enqueue it at another queue. This could repeat *ad infinitum*. Obviously, at some level, such as the first or second, we have to break the chain of events by forcing a LOCK to be implemented by means of a spin lock rather than by enqueueing a task at a semaphore.

In terms of performance, the two alternatives of spin lock and task enqueue have opposite effects on MIPS and MSYPS measures. Task enqueueing tends to increase MIPS available by reassigning idle processors to other useful work. Spin locks tend to decrease MIPS by dedicating potentially useful machine cycles to the effort of repeatedly testing a semaphore. The opposite effect occurs with respect to MSYPS. One effect of task enqueueing is to increase the number and length of critical sections protected by locks. By increasing the number of critical sections, the MSYPS demand is increased. Since only one processor at a time can execute a critical section, by increasing the length of critical sections, presumably because of the various actions required during an ENQUEUE and DEQUEUE, the maximum potential MSYPS rate is decreased.

If a parallel process is limited mainly by MSYPS rather than by MIPS, then the effect of changing spin locks into ENQUEUE/DEQUEUEs will tend to lower throughput. Conversely, if the limitation is a MIPS rather than an MSYPS limitation, then a change from spin locks to ENQUEUE/DEQUEUEs may have the opposite effect. It may lead to higher performance, provided that the ENQUEUE/DEQUEUE overhead is sufficiently low that the system with ENQUEUE/DEQUEUE locks is still MIPS limited rather than MSYPS limited.

Before closing this section, we describe briefly the implementation of UNLOCK because it is very different depending on whether the corresponding LOCK is a spin lock or an ENQUEUE lock. To unlock a spin lock, the owner processor does no more than write a 0 in the semaphore. It is not necessary to do a READ/MODIFY/WRITE to unlock the semaphore.

The performance problem that results when *N* processes are spinning on one semaphore is that the unlocking process is competing with those processes for access to the semaphore and may be delayed an amount of time proportional to *N* while attempting to let another processor pass through the lock. To avoid this problem, the architect can bias the memory system to give priority to a WRITE request over a READ/MODIFY/WRITE request, provided that other rules of arbitration guarantee that every requestor eventually obtains service. A process should not loop endlessly at a lock while other processes receive more than their fair share of service.

If the LOCK operation enqueues idle tasks, then the UNLOCK operation can dequeue a task waiting for that semaphore. The dequeued task can be started after the LOCK without having to test the semaphore, provided that the unlocking process dequeues a task instead of unlocking the semaphore, since a DEQUEUE is the same as an UNLOCK immediately followed by a LOCK. If the UNLOCK operation does not check the queue of tasks waiting at the semaphore, there must be some other mechanism to restart enqueued tasks, for otherwise tasks could wait indefinitely. The dequeueing form of UNLOCK almost certainly requires a READ/MODIFY/WRITE operation instead of a simple WRITE operation because it inspects shared queue pointers, which have to be protected during concurrent updating.

7.2.2 Synchronization with Increment and Decrement

The architect can implement selected instructions that perform READ/MODIFY/WRITE in a way that permits these instructions to perform the same function as Test-and-Set and possibly yield greater functionality as well. Obvious candidates for this purpose are Increment Memory and Decrement Memory, which respectively increment or decrement a designated memory location.

To use these instructions for synchronization, the architect has to implement them in such a way that each instruction "owns" its designated memory cell for the duration of its execution. Once the designated memory cell is accessed by the READ, no other instruction can access that cell until after the modified contents are rewritten to the cell.

A plain Increment or Decrement instruction simply updates an operand and need not perform an uninterruptible READ/MODIFY/WRITE. Such an instruction can be used freely for updating unshared data without regard for correctness of usage in multiprocessor systems.

Only if the instruction is guaranteed to be uninterruptible can it be used as well to update shared data. If an uninterruptible version of the instruction is incorrectly implemented or if a programmer inadvertently uses an interruptible version of the instruction under the mistaken impression that the instruction is uninterruptible, then the instruction works correctly almost all of the time. However, in improbably rare instances, an access by another processor will occur between the READ and the WRITE of the Increment/Decrement instruction, and in these rare instances, a program failure occurs. When used in this manner, the interruptible Increment instruction might well be called "Increment Almost Always," because that is its behavior.

Extensive debugging and program testing is not likely to reveal the existence of a timing hazard in the Increment, and a programmer may be fooled into believing that the program is correct. But a truly correct program must have a truly zero probability of failure, and this requires synchs to be performed by uninterruptible READ/MODIFY/WRITE instructions.

For architectures in which Increment and Decrement are uninterruptible primitive operations, some synchronization functions require fewer instructions with Increment/Decrement than with Test-and-Set. Test-and-Set returns a single bit of information. Increment and Decrement can return the full contents of a memory cell, and the additional bits available can reduce the number of instructions required for synchronization.

For example, consider a shared buffer of length M. Up to M processes can be adding to that buffer concurrently, provided that they operate on separate cells. If M processes are actively adding to a buffer, and one more process requests concurrent access, the M + 1*st* process has to wait. In essence, we need a generalization of a semaphore.

A semaphore as implemented with Test-and-Set permits one process to pass and denies access for subsequent processes until the semaphore is unlocked. This is satisfactory for controlling a buffer of length 1. The generalized semaphore permits up to M processes to pass concurrently and denies access to subsequent processes until one or more processes unlock the semaphore. Each UNLOCK allows one additional process to pass the semaphore.

A very simple means for using Increment and Decrement to implement this form of the semaphore is to start the semaphore with an initial value of M and have each requesting process decrement the semaphore. A processor that sees a nonnegative number after decrementing has access to the buffer. A processor that observes a negative number is blocked from access and should increment the semaphore immediately to reflect the fact that it is not actively working on the buffer. Blocked processes can be enqueued or can retest the semaphore, as discussed earlier for Test-and-Set instructions. A processor that has completed access to a buffer increments the semaphore to indicate that there is room for another process at the buffer. [illegible]

The naive implementation of this form of synchronization exhibits an interesting failure mode known as *livelock*. Program 7.5(a) is a direct implementation of the steps described previously. Buffer access is protected by a decrement of the semaphore. If the result is negative, the semaphore is incremented, and the test repeats to make a spin-lock implementation. If the result is nonnegative, the processor enters the protected section of the program, and exits by incrementing the semaphore.

Program 7.5 Synchronization with and without livelock.

```
while decrement(semaphore) < 0
   do increment(semaphore);
{* Critical Section *}
increment(semaphore);
```

(a) With livelock; and

```
LOOP: while semaphore ≤ 0 do
   if decrement(semaphore) < 0 then
      begin
         increment(semaphore);
         go to LOOP;
      end;
   {* Critical Section *}
   increment(semaphore);
```

(b) Without livelock.

Notes:

1. Instructions **increment** and **decrement** are uninterruptible READ/MODIFY/WRITE instructions.
2. The parameter *semaphore* is a semaphore variable that guards the critical section. [illegible]

The problem is that the system can enter a state in which no useful work is accomplished, yet there are openings available at the buffer—a state of livelock. The "live" in livelock contrasts this state with deadlock, which occurs when a cycle of precedence exists in which A is waiting for B, B is waiting for C, and so forth, with the last item in the cycle waiting for A. Deadlock is "dead" because the state is permanent. The processes within the deadlock cycle cannot end the deadlock unless one or more of them aborts.

Livelock, however, is not inherently persistent. Processors enter a livelock state because of a quirk in timing, and they can leave the livelock state for an active state if timing of events becomes more fortuitous. شانسی

To observe livelock in Program 7.5(a), consider what happens if a huge number of processors issue a decrement to the semaphore immediately after the semaphore reaches a value of 0. The semaphore will then reach a value of $-HUGE$. Will it ever become positive? Not necessarily.

If each of the blocked processors performs an Increment, jump to Retest, and Decrement without interruption, and then turns the semaphore over to the next processor, the semaphore will momentarily change value from $-HUGE$ to $-HUGE + 1$ and then return to $-HUGE$. As the M active processors complete, they will increase the semaphore value to $-HUGE + M$, but this is still negative and will not permit other processors to access the buffer. Hence, useful work is blocked just because of the current order of events. A change in the order of events could result in the semaphore becoming nonnegative, at which point useful work is resumed. از سر گرفتن، دوباره آغاز کردن

Program 7.5(b) shows a mechanism for eliminating the livelock in Program 7.5(a). The trick is to test the semaphore before decrementing. Program 7.5(b) appears to prevent the value of the semaphore from becoming less than -1, but actually it can become very negative.

In the worst possible case a huge number of processors observe a nonnegative value of the semaphore and all proceed to decrement the semaphore, giving it the value of $-HUGE$. Once the processors have decremented the semaphore, incremented it, and are preparing to retest it, no further decrementing is permitted until the value of the semaphore becomes greater than 0.

When the value becomes greater than 0, at least one process is permitted to pass before the value becomes negative again. Hence, useful work continues to be done, although in the worst possible (and highly improbable) case, the average number of active processors is sharply below the available potential.
بشدت. بطور دقیق

7.2.3 Synchronization with Compare-and-Swap

The Compare-and-Swap instruction produces the maximum possible MSYPS rate for a conventional processor because it reduces locked regions of a program to a single instruction—the Compare-and-Swap instruction. A shared datum is locked at the beginning of the instruction, updated during the instruction, and unlocked at the end. This is in contrast to the prior examples, which create a critical section of instructions by manipulating a semaphore before and after the update to a shared datum. The Compare-and-Swap is useful in a limited number of very important circumstances, including the queueing and dequeueing of tasks.

The execution of a Compare-and-Swap is very mysterious at first glance, and only after examining its operation in practice does its power become clear. The Compare-and-Swap operates as defined in Program 7.6. The definition shows that Compare-and-Swap requires two machine registers, one to hold an old value of shared datum, and one to hold a new value.

The objective of updating a shared variable with Compare-and-Swap is to use ordinary instructions to compute the new value of the shared datum without locking it. Then, in one uninterruptible operation, Compare-and-Swap refetches the shared datum, tests to see that its value is unchanged, and if so, performs an update. If the value has changed, the current value is loaded into the register that holds the old value. At this point, the program can recompute a new value and attempt an update with another execution of Compare-and-Swap.

A simple example of the use of Compare-and-Swap is shown in Program 7.7. In this case, the program adds a locally computed increment to a shared variable. Note that the program reads the current value of the variable into

Program 7.6 Compare-and-Swap.

```
Definition: Compare-and-Swap(Address, Reg_old_val, Reg_new_val);
temp := Memory[Address];
if temp = Reg_old_val then
  begin
    Memory[Address] := Reg_new_val;
    Condition_Code := 1;
  end
else
  begin
    Reg_old_val := temp;
    Condition_Code := 0;
  end;
```

Notes:

1. Variable *Address* is a memory address.
2. *Reg_old_val* and *Reg_new_val* are machine registers.
3. The instruction is uninterruptible after it is started.
4. The condition code can be tested after execution is completed to determine if the update took place.

Program 7.7 Updating a shared sum with Compare-and-Swap.

```
  Local_sum := 0;
  for i := 1 to N do
    Local_sum := Local_sum + X[i];
  Reg_old_val := Memory[Address];
LOOP: Reg_new_val := Local_sum + Reg_old_val;
  Compare-and-Swap(Address, Reg_old_val, Reg_new_val);
  if Condition_Code = 0 then go to LOOP;
```

Notes:

1. Variable *Address* is the memory address of a global sum.
2. *Reg_old_val* and *Reg_new_val* are machine registers.
3. The program adds the values of *N* entries of vector *X*, then adds these to the global sum.

Reg_old_val, computes the new value in *Reg_new_val*, and attempts to update the variable with the Compare-and-Swap.

If no conflicts occur during the computation of the new value, the update is successful. If not, the program returns to the loop and computes a new updated value of the sum. Recall from Program 7.6 that Compare-and-Swap loads the current value of the shared variable into *Reg_old_val* in this case, so it is not necessary to read the shared variable again when computing its updated value.

Compare Program 7.7 with our original model of how to update a shared variable with a sequence of LOCK, READ, MODIFY, WRITE, UNLOCK operations. When a LOCK/UNLOCK pair are used, no more than one processor at a time can execute the instructions that perform READ, MODIFY, WRITE.

In Program 7.7 many processors can execute the instructions of this program concurrently, arbitrarily interlacing their access and execution patterns. However, the Compare-and-Swap is uninterruptible. Because many processors can read and write the shared sum, it is possible for the sum to change value between the time a processor reads it at the beginning of Program 7.7 and the time that processor updates it at the Compare-and-Swap. There is no LOCK to prevent such concurrent access.

The key to ensuring correct program behavior is the test made by the Compare-and-Swap. The new value of the shared variable is a function of the old value, and the test ensures that the old value has not changed. If the old value is unchanged, then the new value is correct, and it is stored in the shared variable.

The most valuable application of Compare-and-Swap is for enqueueing and dequeueing without locking. Because queue pointers are shared variables, typical ENQUEUE/DEQUEUE programs lock the queue pointers before changing their values. This creates a multiprocessor bottleneck at the queue routines by limiting the maximum MSYPS rate of a computer system.

Compare-and-Swap provides a means for concurrent updating of queue pointers by limiting the locked segment of code to a single Compare-and-Swap instruction, similar to the way that Program 7.7 limits the locked segment for updating a sum to a single Compare-and-Swap instruction.

The computer literature on this particular application of Compare-and-Swap is rather sparse considering the importance of the idea. Sites [1980] describes the ENQUEUE process, but is not complete because the DEQUEUE process is left as an exercise. Hwang and Briggs [1984] give a rather brief discussion that serves only as an introduction to Compare-and-Swap. Treiber [1986] highlights Compare-and-Swap in more detail in a brief research report. The most complete source of information at this writing is the IBM System/370 Principles of Operations [1983], which gives several examples of correct applications and also shows pitfalls of incorrect use of Compare-and-Swap.

In spite of the apparent simplicity of Program 7.7, Compare-and-Swap is extremely tricky to use correctly. The problem lies in the potentially large number of ways that concurrent execution can occur. After all, the idea of Compare-and-Swap is to foster concurrency. However, when many processors execute the same code concurrently, a variety of events can occur in sequences unforeseen by the programmer, and synchronization can fail. Compare-and-Swap is both one of the most valuable tools for multiprocessor software and one of the most difficult tools to use for that environment.

To show both the power and the danger in the use of Compare-and-Swap, consider the problem of enqueueing data. Figure 7.4 illustrates the data structure for the queue and shows Compare-and-Swap permits queueing to be done with high concurrency. Figure 7.4(a) shows a queue represented as a one-way linked list whose *Head* pointer designates the first item in the queue, the one to be removed next. The *Tail* pointer designates the last item in the queue, the point at which new items are added.

Our objective for concurrent enqueueing is to do the equivalent of the following three-line code segment that places the entry at memory address *Item* at the end of the queue:

```
Memory [Item].Link := nil;
Memory [Tail].Link := Item;
Tail := Item;
```

The notation ".*Link*" denotes a link field of an item in memory. The last two statements in this example have to be executed without interruption because *Tail* is a shared variable that is read, modified, and rewritten.

When the code is executed correctly, the result of inserting one item is as shown in Fig. 7.4(b). However, if Processor 1 and then Processor 2 read the current value of *Tail* at the second statement, then Processor 1 and 2 in that

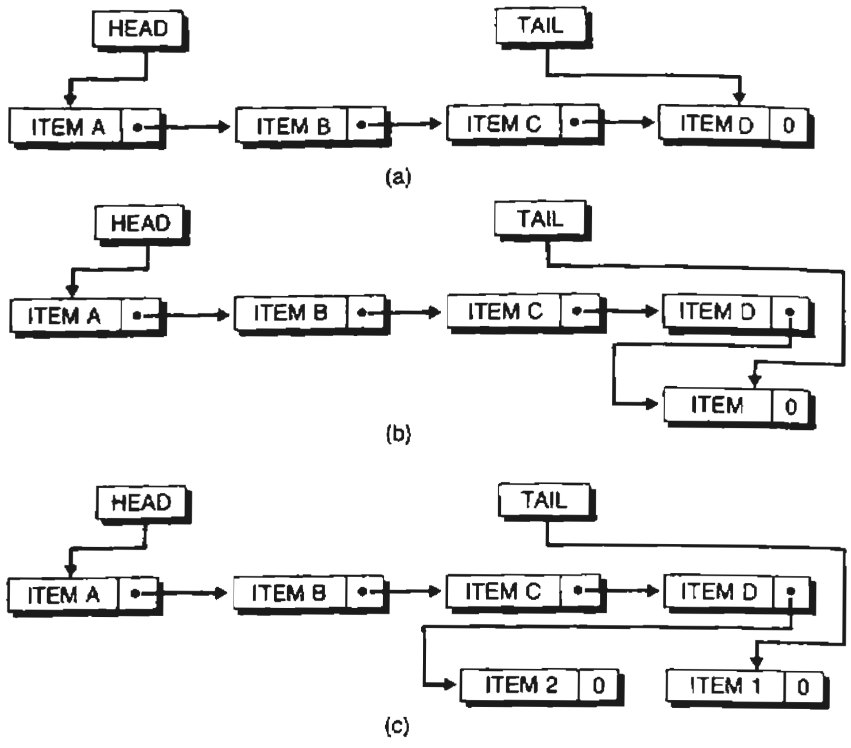

Fig. 7.4 Queues:
(a) A linked-list representation of a queue;
(b) A queue after the insertion of a new item; and
(c) A queue after executing two concurrent insertions without locking. Processor 1 inserts Item 1, and Processor 2 inserts Item 2, with accesses interlaced as described in the text.

order modify the value of *Tail* at the third statement, and then one of the items enqueued will be lost. The pointer to this item will be overwritten. If the last statement is executed first by Processor 2 and then by Processor 1, *Tail* will be left pointing at an item not on the queue. All subsequent items enqueued will be unreachable from the *Head* pointer. This situation is shown in Fig. 7.4(c).

Conventional programming techniques lock this set of statements before they are executed and unlock them when they are completed. A solution based on Compare-and-Swap is shown in Program 7.8. This program avoids the pitfall of an interrupted READ/MODIFY/WRITE. Exactly one processor of a group of concurrently executing processors uses the Compare-and-Swap successfully to read a value of *Tail* and write a pointer to *Item*. This leaves *Tail* pointing to the new *Item*. The former value of *Tail*, now in the register *Reg__Tail*, points to the former end of the queue. The queue is extended by linking that entry to *Item*.

If a Compare-and-Swap fails, the processor repeats with the new value of *Tail* that was loaded into *Reg__Tail* by the Compare-and-Swap. The net effect of Compare-and-Swap is to guarantee that the values stored in *Tail* and in *Memory*[*Tail*].*Link* are consistent.

By various arguments we can show that Program 7.8 is correct for concurrent ENQUEUE operations. However, the program as written does not treat empty lists in full detail, nor is it correct if ENQUEUE operations occur concurrently with DEQUEUE operations. These additional considerations greatly complicate matters. Compare-and-Swap is extremely difficult to use correctly as complexity grows, and its use is prone to very subtle errors that may never be detected.

Consider, for example, Program 7.8 when execution reaches the Compare-and-Swap. This statement relies on the fact that if *Reg__Tail* = *Tail*, then no other concurrent ENQUEUEs have updated *Tail* since it was last read from memory. If we allow concurrent DEQUEUEs as well as concurrent ENQUEUEs, this may not be the case. A DEQUEUE could have removed the item at *Reg__Tail* from the queue, and subsequently, an ENQUEUE could have reached a Compare-and-Swap to restore this item to the queue. This would leave *Tail* at its former value, a value equal to the contents of *Reg__Tail*, and we have reached a condition at which two different processors will attempt to update *Memory*[*Tail*].*Link* with different addresses.

We call this failure mode the "A-B-A problem" because it occurs when a variable takes on sequence of values such as *A*, *B*, then *A*. A Compare-and-Swap tests the value *A*, and presumes that the value was held continuously if it is there currently. The Compare-and-Swap is unable to detect that a change

Program 7.8 Enqueueing an item with Compare-and-Swap.

```
    Memory[Item].Link := nil;
        {* Initialize Item for insertion at end of queue *}
    Reg__Tail := Tail; {* Read Tail to a register *}
LOOP: Compare-and-Swap (Tail, Reg__Tail, Item);
    if Condition__Code = 0 then go to LOOP;
        {* Loop back on failure of Compare-and-Swap *}
    Memory[Reg__Tail].Link := Item;
```

Notes:

1. This program is correct for concurrent ENQUEUEs.
2. The program as written here may fail if DEQUEUEs and ENQUEUEs can execute concurrently.
3. Dequeueing may require additional tests, depending on the handling of empty lists.

of state has occurred. Although the A-B-A problem is unlikely to occur, it is not impossible.

For Program 7.8 the consequence of the A-B-A problem is program failure. The fact that *Tail* equals *Reg_Tail* is treated as if *Tail* has not changed since it was last read. However, this inference is incorrect, and any sequence of events that leaves *Tail* in its original state can potentially lead to failure of Program 7.8. Since concurrent ENQUEUEs by themselves cannot restore the value of *Tail*, Program 7.8 is safe for concurrent ENQUEUEs.

A practical solution to improving the safety of Compare-and-Swap is outlined in the IBM System/370 Principles of Operations [1983]. The idea is to extend the Compare-and-Swap to deal with two variables rather than one. The two variables must be contiguous so that they can be fetched and rewritten with one READ and WRITE.

Program 7.9 illustrates how this extension improves the code reliability. In this program, *Tail* is concatenated with a variable *Count*. The current value of the *Tail/Count* pair is copied to local registers. Just prior to the Double Compare-and-Swap, the local copy of *Count* is incremented and moved to the register *New Count*. The Double Compare-and-Swap verifies that the *Tail/Count* pair has not changed, and it updates this pair of values with the pair *Item/New Count*.

Since each successful execution of Compare-and-Swap updates both *Tail* and

Program 7.9 Enqueueing an item with Double Compare-and-Swap.

```
   Memory[Item].Link := nil;
      {* Initialize Item for insertion at end of queue *}
   Reg_Tail&Reg_Count := Tail&Count;
      {* Read double-variable Tail and Count to two registers *}
LOOP: New_Count := Reg_Count + 1;
      {* Prepare to update Count *}
   Double Compare-and-Swap (Tail&Count, Reg_Tail&Reg_Count, Item&New_Count);
   if Condition_Code = 0 then go to LOOP;
   Memory[Reg_Tail].Link := Item;
```

Notes:

1. The notation *Tail&Count* designates two variables stored contiguously or two contiguous registers that are accessed by a single double-length operation.
2. Double Compare-and-Swap reads a double-length operand from *Memory*[*Tail*], compares this to the double-length operand *Reg_Tail* and *Reg_Count*, and updates *Tail* with the double-length operand *Item* and *New_Count* if the equality comparison is satisfied. *Reg_Tail* and *Reg_Count* are updated if the equality comparison fails.
3. The program as written here may fail if *Count* is incremented a sufficient number of times to overflow back to its original value, and *Tail* is left in its original state.

Count, if *Tail* is changed and restored by concurrent queue operations, then the new value of *Count* would show that other queue operations have taken place or are in progress concurrently. This forces an unsuccessful Compare-and-Swap, which in turn causes a loop to occur and prevents an erroneous update. An update takes place only if both *Tail* and *Count* have not changed.

The sustained value of *Count* is intended to signify that no other concurrent operations are in the process of manipulating *Tail*. In the System/370 architecture, *Count* returns to its former value after no sooner than 4 billion operations. Consequently, Program 7.9 has a highly improbable failure mode in which a failure occurs if a process is suspended at a Double Compare-and-Swap while other processors increment *Count* 4 billion times and leave *Tail* in its original state.

Program 7.9 does not indicate the proper treatment of concurrent ENQUEUEs and DEQUEUEs. After a successful update of *Tail*, the ENQUEUE process must update the link field of the predecessor of the new item. In Programs 7.8 and 7.9, this is done by a simple assignment statement. To permit concurrent DEQUEUEs, the update should be done by using a Compare-and-Swap to discover if the prior link were 0, which is the correct condition for the last item in the queue prior to an insert. If the DEQUEUE process removes the item before the ENQUEUE process sets the link, we assume that the DEQUEUE process detects this condition, and stores a special nonzero flag in the link field to indicate special handling is required. The ENQUEUE should use a Compare-and-Swap rather than a simple assignment to update the link in Program 7.9 because a DEQUEUE process may be conditionally storing a special flag concurrently. The precise details of the correct ENQUEUE and DEQUEUE process provide an interesting challenge to the reader, and is the subject of one of the exercises in this chapter.

To summarize the characteristics of Compare-and-Swap synchronization, it is extremely efficient, and highly desirable to use. However, it is very dangerous and subject to subtle failure modes. It has to be used carefully and by experienced programmers.

7.2.4 READ/MODIFY/WRITE with Reservations

The Compare-and-Swap instruction can be viewed as the WRITE function of a very long READ/MODIFY/WRITE instruction whose READ occurs many instructions earlier. An atomic READ/MODIFY/WRITE is executed with no intervening operations. When paired with a READ, the Compare-and-Swap is a nonatomic READ/MODIFY/WRITE. Consider the following sequence of events:

1. *READ*: *X* is copied to a register by READ *X*.
2. *MODIFY*: An updated value of *X* is computed and stored in a register as the value *X'*.

3. *WRITE*: A Compare-and-Swap writes back the value of X' into the memory location X, provided that the value X is unchanged.

Other instructions can intervene between the READ and the WRITE, which is permitted, provided that they do not alter the value of X in memory. (Although we interpret success of Compare-and-Swap to indicate that X was unchanged between the READ and Compare-and-Swap, the Compare-and-Swap succeeds if X takes on a sequence of values A, B, A between the READ and the Compare-and-Swap as discussed in the previous section.)

The advantage of this approach is that other processors can continue to perform useful work while a processor is engaged in the updating of X. If the process updating X suffers a significant delay, say from a page fault or from being swapped out of memory, other processes can still gain access to X and attempt updates that may succeed. Compare this to a critical section in which X is locked during an update. If a process is suspended, interrupted, or swapped out of memory while it holds locks, no other process that needs the locked data can make progress. Hence, the Compare-and-Swap is an alternative way of performing an update that takes many instructions without holding a lock on the shared data while the update is performed.

The A-B-A problem is a significant nuisance for the Compare-and-Swap, and it would be interesting to consider alternate ways of doing READ/MODIFY/WRITE nonatomically. But we must guarantee that the WRITE occurs only if the variable is not modified by another processor or by another process on the same processor between the READ and the WRITE functions. An approach based on reservations can be used to solve this problem.

The key idea is that a multiprocessor with a coherent cache has most of the necessary hardware in place to support this facility. Recall that the cache-coherence protocols require that a potential writer of data have a write privilege. For each storage location in main memory, there is at most one processor that has write privilege for that location at any time. We can use this state information to construct the functions we need for a nonatomic READ/MODIFY/WRITE.

The READ will be done by an instruction that we call READ_AND_RESERVE. From the program's point of view it accesses the shared variable. But it also creates a reservation for the variable. The reservation could be a single bit set in the cache line, or could be a dedicated machine register to record the existence of the reservation. The reservation remains active until any processor writes to the shared variable. The processor must be able to recognize during a cache access when a variable is reserved, and the processor occasionally must be able to find and cancel the active reservations that it holds.

The WRITE will be done by an instruction that is a conditional store, WRITE_IF_RESERVED. The WRITE_IF_RESERVED instruction stores the contents of a machine register at a location in memory, if the processor currently holds a reservation for that location. If the reservation has been abandoned, then the store fails. The instruction returns a condition code that can be tested to indicate if the store succeeded or failed.

The idea is that the reservation remains in force between the READ_AND_RESERVE and the WRITE_IF_RESERVED as long as the reserved variable is unchanged. If another processor or process on the same processor alters the reserved variable, the reservation has to be removed.

All cache-coherence protocols have hardware support to request, transfer, and abandon write privilege because all protocols assure that at most one processor has write privilege for any particular memory location. When another processor needs to update a shared variable, it will request write privilege if it does not already have this privilege or it will send out invalidate messages if it does. Whichever message is sent, the message cancels reservations at the receiving processors. If the reservation is held in a cache in a status bit in the reserved line, the status bit is reset if the line is updated. If the line is invalidated, the status bit ceases to exist, which is an indication that no reservation is active. If the reservation is held in a dedicated register, the register is marked empty.

This is an elegant solution to the A-B-A problem. The simplicity of implementation is due to the reservation approach being quite compatible with the cache-coherence protocol. The communication for managing write operations already exists among the processors. If the communication were not there already, the reservation technique would be a costly addition to an architecture.

There is a small amount of additional complexity required to handle access to a shared variable by several different processes that execute on the same processor. If one process, say Process A, holds a reservation, and a context swap shifts to Process B, a process that also holds a current reservation for the same variable, B may execute a WRITE_IF_RESERVED instruction that succeeds when in fact the reservation was for the value read by A, not the value read by B. So the reservations for different processes need to be distinguished somehow, or we must force processes to abandon all of their pending reservations when a context swap occurs. If the latter approach is used, the processor must be able to find and remove all pending reservations and to do this as quickly as possible. To cancel quickly, it is advantageous to hold reservations in dedicated registers.

When reservations are available, all of the READ/MODIFY/WRITE instructions can be implemented nonatomically with reservations. For example, consider Test-and-Set:

```
  REG[1] := 1;
LOOP
  REG[2] := READ_AND_RESERVE (semaphore_Z);
  WRITE_IF_RESERVED (REG[1], semaphore_Z);
  if unsuccessful go to LOOP; {* WRITE Failed, try again *}
     {* Test prior value of semaphore and return condition code *}
  return REG[2] = 0;
```

We do not expect to go around the loop at all, but rarely the loop may be taken once, possibly twice. The READ and WRITE are only one instruction apart, which makes the loss of a reservation very unlikely. When the READ and WRITE are separated by many instructions, there is a greater probability of losing a reservation in the interval between them. Hence, the interval should be kept as short as possible to increase the probability of success.

Since both READ_AND_RESERVE and WRITE_IF_RESERVED are synchronizing instructions, they must be implemented to be compatible with the consistency model of the computer system. That is, if the model uses release consistency, the WRITE_IF_RESERVED should behave like a RELEASE and the READ_AND_RESERVE should behave like an ACQUIRE.

Compare-and-Swap algorithms are greatly simplified when implemented with reservation instructions since they do not have to contain the additional code to protect against the A-B-A problem. The reservation system also simplifies the implementation of the storage subsystem. Conventional machines implement three types of storage operations: READ, WRITE, and READ/MODIFY/WRITE. The latter are sufficiently different to add cost and complexity. By using reservation instructions to implement the READ/MODIFY/WRITE functions, the storage subsystem operations can be reduced to just READ and WRITE. Whether or not reservation instructions are used, the storage subsystem should also distinguish between synchronizing instructions and ordinary instructions to enable programs to remove problems caused by competing accesses.

7.2.5 Synchronization with Fetch-and-Add

The three synchronization methods discussed thus far have in common the property that they are serial methods. No more than one processor at a time can execute the READ/MODIFY/WRITE operation embedded in them.

The Fetch-and-Add operation is different—it is truly parallel. Conceivably, all *N* processors in a multiprocessor can execute a Fetch-and-Add instruction simultaneously, provided that all processors update the same variable. Fetch-and-Add operations executed on different variables may have to be done sequentially if those variables reside in the same memory or share access circuitry of some other form.

The instruction Fetch-and-Add(*Sum, Increment*) provides for adding an increment to a shared sum, and the addition is done in parallel as explained earlier. No locking and unlocking is required, nor is a retry test and loop required as with Compare-and-Swap.

In terms of performance, the Compare-and-Swap is as efficient or more efficient than the Fetch-and-Add if on the average only one processor at a time requests an update of *Sum*. This is because Compare-and-Swap is not burdened by delays by network access introduced by the hardware implementation of

Fetch-and-Add. However, when the update becomes a bottleneck to the extent that 10 or 100 requests for access are active concurrently, Fetch-and-Add is far faster than Compare-and-Swap because it can honor all the requests simultaneously.

For systems with relatively few processors, Compare-and-Swap is the better approach. As the processors increase, Fetch-and-Add provides potential performance improvement not available with Compare-and-Swap. Fetch-and-Add becomes more attractive as the number of processors increases, but whether or not Fetch-and-Add is cost-effective is still a matter of research interest. Its implementation cost is high, and its potential is limited to simultaneous access of the same shared variable by all contending processors. It provides no help for contention produced by concurrent accesses to different variables in the same memory.

For large values of *N*, for example 1000 to 10,000, Fetch-and-Add or an equivalent mechanism for parallel synchronization is a practical necessity. Without such a mechanism the MSYPS limit will severely impair performance in a 1000-processor system. In 10,000-processor systems, other system bottlenecks may be so severe that Fetch-and-Add by itself may not be sufficient to produce acceptable performance.

To show Fetch-and-Add at its best, let us reconsider the problem of enqueueing and dequeueing items on a shared queue. The Compare-and-Swap approach is pointer oriented, that is, the links are treated as addresses, and the algorithm builds linked lists.

Fetch-and-Add, however, is best used for counters rather than pointers, where counters are variables that are manipulated by addition and subtraction. The result of a sequence of counting operations is not sensitive to the order in which increments and decrements are applied, which is desirable for Fetch-and-Add because concurrent executions receive a set of results that represent some arbitrary ordering of the individual summations. We want to create algorithms for which all of the arbitrary orderings are consistent with correct execution of the algorithm. Consequently, the most appropriate implementation of ENQUEUE with Fetch-and-Add is to use a counter-based implementation.

The basic idea is to use a counter, *Tail*, that is incremented by ENQUEUE. The value of *Tail* is the offset in the queue of the next insertion point. A simple and incomplete implementation of ENQUEUE with Fetch-and-Add is

```
Procedure Enqueue(Item, Queue);
  begin Place := Fetch-and-Add(Tail,1);
    Queue[Place] := Item;
  end; {* Enqueue *}
```

The Fetch-and-Add increases *Tail* and returns the value of *Tail* before the increment. This value is used as the offset in the queue for inserting an item. If the Fetch-and-Add is executed simultaneously by several processors, *Tail* receives the sum of the increments, and each processor receives a different value for *Place*, so each processor has a unique position for queue insertion.

This is the basic idea of enqueueing with Fetch-and-Add, but the full implementation becomes very complex because of a variety of conditions that have to be satisfied. Among the conditions are:

1. The queue should be circular, so *Tail* should be set to a base value of 0 when it exceeds the length of the *Queue* vector.
2. The total number of active entries in the queue cannot exceed the length of the queue vector.
3. The DEQUEUE operation should permit parallel removal entries from the queue.
4. The DEQUEUE operation should not permit a dequeue to succeed on an empty queue.
5. Both ENQUEUE and DEQUEUE should be safe from livelock.

Two implementations of ENQUEUE/DEQUEUE with Fetch-and-Add appear in Gottlieb *et al.* [1983] and Stone [1984]. Both solutions are too complex to reproduce in this text. However, the implementations illustrate general principles worth discussing here.

If we use variables *Tail* and *Head*, respectively, to control the insertion and deletion points in a queue, then the number of items in a queue is the difference between *Tail* and *Head*. However, because both *Tail* and *Head* are reset to 0 when they exceed the length of the queue, the difference in their values is the number of active elements modulo the length of the queue, so finding the number of active elements from the values of *Head* and *Tail* is rather tricky. It is much easier instead to maintain a separate variable *Count* that gives the current number of active elements. ENQUEUE and DEQUEUE operate on this variable with Fetch-and-Add with increments of $+1$ and -1, respectively. The value returned by Fetch-and-Add can be used to control actions on queue overflow and underflow.

To prevent livelock, ENQUEUE should first test *Count* before incrementing it, and DEQUEUE should test *Count* before decrementing it. The queue full and queue empty conditions that cause processors to loop back to retry their operations should loop back to the test of *Count* in a manner similar to the way that livelock is treated with the Increment and Decrement instructions. In this way processors remain at the outermost test and are prevented from further incrementing or decrementing until *Count* reaches a safe value.

To handle the queue circularity, when a Fetch-and-Add increments *Head* beyond the end of the queue, the set of processors making concurrent access to *Head* will discover its value to be less, equal to, or greater than the queue length. The processors that receive legal values for *Head* simply continue. The processors that discover values beyond the end of the queue abort their activity by decrementing *Head* and return to a place earlier in the program to request a spot in the queue again. Eventually *Head* will return to the least illegal value.

The processor that decrements *Head* to this value decrements *Head* again by the length of the queue and thereby resets *Head* to start at the beginning of the *Queue* vector. Livelock prevention tests have to protect *Head* from livelock during the incrementing and decrementing that occur in this process.

The full algorithm for ENQUEUE/DEQUEUE

- Manipulates *Count*, *Head*, and *Tail*;
- Handles queue circularity, queue empty, queue full; and
- Protects from processing livelock.

Working out the details of the algorithm is very instructive and shows the complexity of synchronization with Fetch-and-Add.

We stated that Compare-and-Swap is difficult to use correctly, but Fetch-and-Add is far more difficult to use. Compare-and-Swap is subject to subtle failures from concurrency before and after it is executed. Because it forces serial behavior when it is executed, some simplification is achieved when verifying the correctness of Compare-and-Swap algorithms. But Fetch-and-Add supports all of the concurrency of Compare-and-Swap and more.

The fact that many processors can perform Fetch-and-Add concurrently on the same datum greatly increases the number of possible outcomes to consider and makes verification extremely difficult. Obviously, Fetch-and-Add has to be used very carefully by experienced programmers. Fetch-and-Add synchronization will probably be used mostly through library calls rather than individually programmed statements because most programmers are not likely to be able to create correct, efficient programs based on Fetch-and-Add.

Although Fetch-and-Add points the way to break the MSYPS bottleneck, the implementation of Fetch-and-Add in a multilevel interconnection is expensive and its use in programs is difficult and error-prone. Is there any effective alternative? Indeed, there are several less powerful, but far less expensive techniques to implement the most useful feature of Fetch-and-Add—the ability to parallelize synchs. These are treated in the next section.

7.2.6 Other Architectural Support for Parallel Synchronization

This section discusses low-cost implementations of a collection of alternatives to Fetch-and-Add, none of which has the full power and generality of Fetch-and-Add. The reason for considering alternatives is that Fetch-and-Add does not make efficient use of the hardware required for its realization. The total number of nodes in a switching network that can accept N concurrent requests is at least $O(N)$ if the network is a simple tree, and is $O(N \log N)$ if the network is a full shuffle-exchange network as proposed by Gottlieb *et al.* [1983]. However, in actual practice the majority of the combining is performed at the nodes at the root of the tree centered on a hot spot, and very little combining is performed at the leaves. Consequently, the cost of the network in components grows at least linearly in the number of processors, and the delay experienced grows as the depth of the network times a large constant to reflect the delay per combining node in the network. Yet, on the average, only very few nodes in the network actually do useful work in practical cases.

When a combining network is working at peak capacity, all of its leaves receive combinable requests simultaneously, and these all combine stage-by-stage to produce a single request at the hot-spot root. If this behavior were typical of every machine cycle, then a combining network would produce a performance commensurate with its cost. What happens in actual practice is that the combinable requests are received over a period of time. If two combinable requests enter a node on different inputs in the same cycle, they are combined into a single request. If they enter on different cycles, they are forwarded sequentially toward the root of the tree on the same path. Thus, there is a window of time during which two requests can combine at a node, and if they miss that window they will not combine there. Each node can include some buffering to enlarge the window of combination to greater than a single cycle, but the effect of buffering is to add delay and cost in each node.

In the exercises for this chapter is one that reveals that each request is most likely to combine with a request at the root of the tree because half of the possible requests join with it there. Half as many requests join with a request at the second level in the tree, and half as many in the next level, and so on. When requests arrive at a combining network over a period of time, with high probability they pass through the first few levels of the tree without combining, and eventually combine near or at the root if the arrival rate is high enough to produce one or more requests per cycle at the network inputs. If k out of N requests arrive on the average per cycle, the requests will tend to saturate the $\log k$ levels of the combining network at the root, and relatively few requests will be combined at other levels.

Consequently, the peak rate of combining supportable by a combining network is far greater than the actual rate that the network has to support. An effective compromise for the computer architect is to put full combining only in a few levels of a combining network, and to make the remainder of the nodes the same as transmission nodes in a conventional multilevel interconnection network.

Given that a combining network may be more powerful than what is actually required, what less powerful functions can be implemented to produce the capability that we actually need? A good candidate is to implement the synchronization functions on a global bus that visits all processors. The reason that this is attractive is that synchronization by itself does not demand high bandwidth. A synchronization message can be as short as one bit. The bit of a synchronization changes rather slowly in time, possibly once every 10,000 to 100,000 instructions. We can afford to deliver the bit a few cycles late because of low bandwidth on the delivery system, provided that the lateness is a constant delay or grows very slowly with the number of processors in the system.

We propose a sequence of bus-based synchronization techniques, each more powerful than its predecessor. They are:

1. Barrier synchronization.
2. Multiple barriers.
3. Find the maximum.
4. Fetch-and-increment.

The first of these, barrier synchronization, has been implemented successfully on the PAX computers by Hoshino [1989]. Each processor sets a single bit to indicate when it has arrived at a barrier. The collection of bits is brought to an AND gate and an OR gate, each of which has one input per processor. The outputs of both gates are bused to all processors. Thus every processor can determine when all processors have reached the barrier, when none have reached it, or when some but not all have reached it. The number of synchs per cycle in this machine grows almost linearly with N. (The growth would be linear if a change in a bit could be propagated in a single cycle regardless of N. In practice, the propagation time grows slowly as N grows, and is $O(\log N)$ with a very small constant coefficient.) The delay in performance caused by a barrier is measured by the number of cycles after the last processor arrives at the barrier before the collection of processors can begin new tasks. This is a few machine cycles at most, even in a multiprocessor with 1000 processors.

A practical implementation of a fast barrier was patented by Thompson [1985] and is shown in Fig. 7.5. Thompson proposed to use an adder with fast-carry lookahead to implement a barrier. In this case, the adder is capable of adding two 4-bit numbers and an incoming carry to produce a sum output and a carry output. The sum output is ignored for the barrier function, and the carry output is fed back to the four inputs of one 4-bit operand. This configuration can synchronize five different processors. Each processor is assigned one of the five available inputs, either one of the four inputs for the 4-bit operand or the input for the carry in.

We assume that we start in a state in which no processors are at the barrier. All processors place a 0 on their respective inputs, and the carry out produced by the device is 0. The carry out remains at 0 until all processors reach the barrier. At this point it becomes 1, and stays at 1 until all processors signify that they have observed the synchronization by removing their 1 bits. When all processors have left the barrier, the carry out drops to 0. If each processor uses

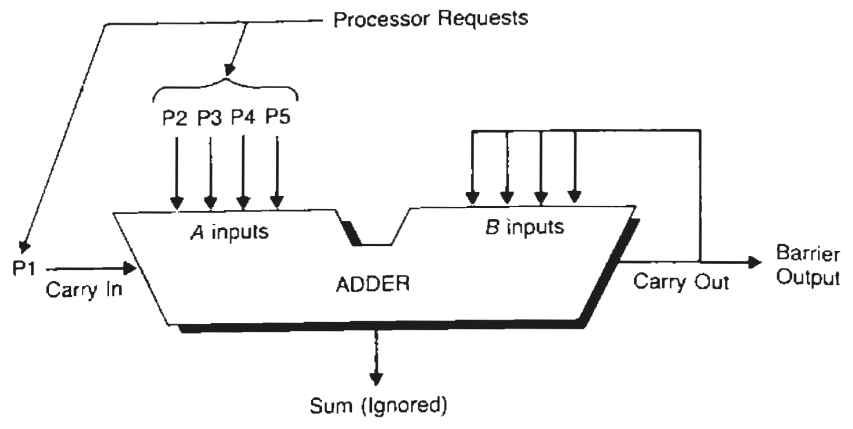

Fig. 7.5 The Thompson barrier.

a rule that it places a 1 on the barrier only if the present barrier value is 0, then the barrier can be safely reused in a program. Thus, Thompson's barrier supports both the reuse of the barrier and the fast implementation.

In a 1000-processor multiprocessor, a single barrier is inadequate for synchronization. But since many processors use an active barrier, in actual circumstances perhaps only 32 or 64 distinct barriers are sufficient to support 1000 processors. The technology to build 32 or 64 Thompson barriers with 1000 inputs each is much less demanding than the technology required to put 1000 processors and memories together, so that we can conceive of a multiprocessor supported by a collection of addressable barriers. However, the interconnections to these barriers present an interesting challenge to the architect.

The scheme illustrated in Fig. 7.6 indicates how one might take advantage of the low bandwidth on the barriers to reduce the interconnection complexity. It is based on work by Heidelberger, Rathi, and Stone [1989]. The device shown in Fig. 7.6 contains all of the addressable barriers for some subset of processors, and it produces summary data on a few output lines that are forwarded to a similar chip whose function is to synchronize a different subset of processors. Each processor is attached to one chip through one input line and to the output bus, and possibly to one dedicated output line per processor. A processor signals its intentions to a chip sequentially by giving the address of a barrier and the value of the bit to set at the barrier. The chip also recognizes special codes for initialization of a barrier, and for masking in or out the processors that are not participating at a barrier. Since one barrier chip may be limited by input/output pins to handling requests from some fixed number of processors, the barrier chip outputs have to be combined together to produce a single global bus output

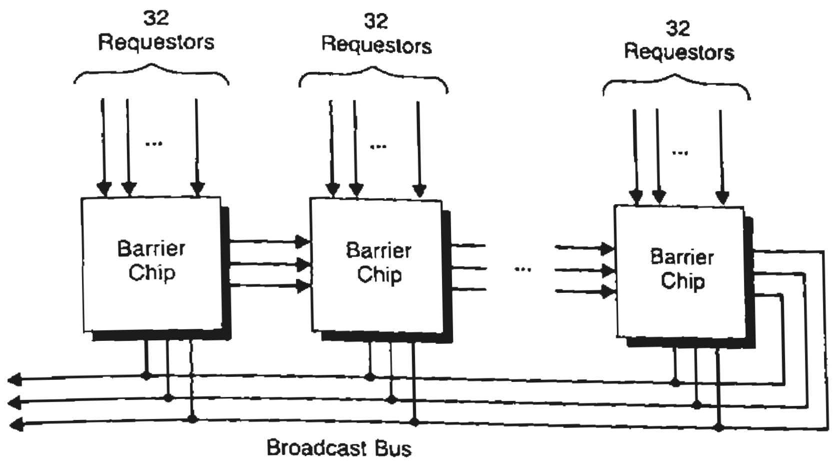

Fig. 7.6 A VLSI implementation of a set of barriers.

that is observed by all processors and by all chips. In general, each chip output and the global bus output are an indication of the states of the various barriers. The output bus may have many lines, one for each barrier, if technology provides this capability at reasonable cost. If this is not feasible, the output bus can signal when any barrier changes state by signaling the address of the barrier and its new state. Since state changes of barriers are quite rare, a bus with a single conductor or a few conductors may have sufficient bandwidth to satisfy the requirements for a large multiprocessor, and a few conductors should provide ample bandwidth to meet peak requirements.

Given that metal interconnections are limited in bandwidth, and are bulky and costly, an all optical barrier may one day be practical and be preferred to a device based on the Thompson barrier. An optical barrier can be constructed by using two wavelengths, λ_{busy} and λ_{done}, to signify, respectively, that a processor is busy before reaching a barrier or has reached the barrier [Green and Stone 1990]. We presume that every processor can produce illumination on one wavelength or the other, and that the illumination can be amplified and bused to all other processors. Each processor operates two receivers, one sensitive to λ_{busy} and the other sensitive to λ_{done}. The illumination from all processors is combined at or before it reaches a receiver, so that each receiver sees a composite signal. If a receiver detects energy at its tuned wavelength, it concludes that some processor is in the state associated with that wavelength, either *busy* or *done*. The barrier is unused when no processor is busy and no processor is done. It is an active barrier when at least one processor is busy. Processors can move past the barrier when no processor is busy. Before a barrier can be reused, each processor must verify that no processor is signaling "done." Only when all processors have left the barrier, can any processor safely signal reuse the barrier for a new cycle by signaling "busy." This implementation is an optical analog of the AND and OR gates used by Hoshino in PAX.

The optical system can support multiple barriers on a common interconnection system by using pairs of wavelengths for each distinct barrier. Because each individual barrier requires so little of the available bandwidth, there is substantial available bandwidth to multiplex many barriers together on one optical interconnection system.

Since optical technology for the barrier application is still in its infancy, we cannot claim that the implementation described here is feasible today or will be the preferred embodiment when and if optical technology can support barriers. Nevertheless, the discussion indicates how new technology may alter the approaches that we take to solve specific problems.

The next function of interest is the ability to find the maximum value of a set of values held in distinct processors. A classic method to find the maximum is for each processor to gain exclusive access to a single global shared variable, and to update the variable if the local value is greater than the global value. Each processor then proceeds to a barrier where it waits until all updates have been done. Then the global value is known to be the maximum of all values.

This process is rather inefficient to perform in a highly parallel multiprocessor because of the serial bottleneck it creates at the global variable. A FETCH-and-MAX operation is an effective, but costly solution. Recursive doubling as described in Chapter 4 is also effective, but the delay in the process requires the latest participating processor to make O(log *N*) remote comparisons before reaching the barrier. This determines how soon the processors can be released from the barrier as a function of the time when the latest processor initiates its computation of the maximum. We may be able to reduce this number, or reduce the cost of a comparison.

An efficient solution lies in the use of a broadcast bus of low bandwidth that visits all processors. Each processor attempts to gain access to the broadcast bus, compare its local value to a global value, update the global value if necessary, and broadcast the new value of the global variable. If, while a processor's request for the bus is pending, the processor observes a higher global value, it removes its request and moves to the barrier. If all processors are vying for the bus when the latest participating processor requests a bus transaction, approximately O(log *N*) bus transactions on the average will take place before all processors reach the barrier. If only *k* processors are vying for the bus at this point, then only O(log *k*) bus transactions are necessary. This is a small reduction if all processors tend to initiate the computation of the maximum in a short interval of time. But the reduction is quite useful and worthwhile when a few stragglers tend to arrive very late, in which case, *k* may be only 1 or 2 in such situations.

What is interesting about this form of solution is the fact that the solution is compatible with the barrier chip solution of Fig. 7.6. The device in that figure needs to be only slightly more versatile to support both barriers and the maximum operation, but the interconnections shown are sufficient to implement both functions.

The last function of interest is Fetch-and-Increment. This is the same as Fetch-and-Add except that the value added has to be +1. We can also support a Fetch-and-Decrement with a minor embellishment of the basic idea. Thus, a processor can add or subtract 1 from a global counter, and perform this in parallel with all other processors. This function is sufficient for performing enqueueing and dequeueing operations without enforcing a strict serialization.

The basic implementation of this idea was discovered independently by Heidelberger, Rathi, and Stone [1990] and Sohi, Goodman, and J. E. Smith [1989]. It uses a bus much like the bus described above for computing the global maximum. All processors that wish to perform Fetch-and-Increment request access to a global bus. When any one processor is granted access, all other processors observe the address that is to be incremented. Then all requestors for that address respond together with the original requestor in a portion of the same bus cycle reserved for responses. For *N* responders, the bus should have *N* distinct data conductors. Each responder places a 1 on a conductor dedicated to that responder. If a processor is not an active responder because it does not have a Fetch-and-Increment pending, then its corresponding conductor carries the logic value 0. Since all processors see all data wires, each processor can tell the total increment applied to the global variable and each can tell its priority with respect to all other processors. For example, if Processor 5 sees that Processor 2 and 3 have responded, then Processor 5's request is third in line. If the request is for a queue entry, then Processor 5 can immediately access the third entry of a queue.

The scheme has to provide a means to update the global variable with the sum of the increments of the requestors. A suitable protocol is to assign the update task to the processor with highest or lowest priority. Also, all requestors should see the current value of the global variable in order to compute their local variant of the global value. The current value can be transmitted on the same bus on a different part of the same cycle. This scheme is also compatible with the implementation described in Fig. 7.6, provided that the bus has a number of wires equal to the number of processors. By multiplexing and coding responses, it is possible to reduce the number of physical conductors. In order to achieve the best performance at reasonable cost, the number of distinct conductors should be roughly equal to the expected number of active requestors on any cycle.

This brings us to the end of the discussion of synchronization techniques. The next section revisits cache coherence and describes why cache coherence and synchronization are alternative solutions to the same problem.

7.2.7 Cache Coherence versus Synchronization

The cache-coherence protocols described in Chapter 6 assure that multiprocessor programs can synchronize correctly. Consider, for example, the software implementation of a barrier. A simple scheme is to use a counter initialized to N, and to have each of N processors decrement the counter as they reach the barrier. If the decrement operation is an uninterruptible READ/MODIFY/WRITE operation, then each processor need do nothing more than decrement the barrier and test it continually until the barrier reaches zero. No lock, Fetch-and-Add, or other specialized technique is required to implement the barrier. The cache-coherence protocol assures that each processor will eventually see the zero value of the barrier variable, and the barrier variable will become zero if and only if all N processors have reached the barrier.

If this is the case, then why is there a special requirement for synchronization hardware such as the combining network or the Thompson barrier? The issue is the implementation of the cache-coherence protocol. Bus-based protocols are limited to one bus transaction per cycle. If at most one synchronization can be done per bus transaction, then an MSYPS bottleneck exists. For multiprocessors with hundreds or thousands of processors, a cache-coherence protocol is not likely to be bus based, and its implementation is an interesting question in itself.

The point of this section is to illustrate that the synchronization techniques explored thus far may provide a substantial part of the cache-coherence function, and conversely, a cache-coherence protocol can provide a reasonable means for implementing synchronization primitives. For large numbers of processors, a dedicated synchronization subsystem may be preferred, and for a small number of processors, a bus-based cache coherence protocol may be preferred. These comments are illustrative of possible choices, and actual decisions must account for the characteristics of the applications.

We proceed by illustrating how a cache-coherence protocol can assure the correctness and efficiency of a barrier synchronization. For the barrier, we will decrement a counter as described above. The cache-coherence protocol is bus-based, and assumes that all processors observe all bus transactions. In Chapter 6, we indicated that we have some choice of protocol. To conserve the use of the bus, recall that it is not necessary to broadcast the update of a cached variable if that variable is held exclusively. To obtain a variable exclusively, it is sufficient to broadcast a cache-invalidate command to all other processors when a processor with permission to update the variable actually performs the update. Let's call this protocol the Write-Invalidate Protocol.

An alternative protocol is to broadcast the updated value of the variable at the time of its update. Any other processor that holds that variable in cache, can overwrite the local value with the updated value. Of course, when this occurs, the variable is not held exclusively by any processor, and subsequent updates have to be broadcast. We call this protocol the Write-Update protocol.

Yet another possibility is for each processor to load into its cache the value of a broadcast update, regardless of whether or not it currently has a copy of that variable. This is contrary to what seems to be reasonable, but it is worthwhile to include it for comparison purposes. We call this protocol the Write-Load protocol.

In terms of efficiency, our expectations are that the three protocols fall in the order below from most efficient to least efficient:

1. Write Invalidate,
2. Write Update,
3. Write Load.

The reasoning behind this ordering is that the Write Invalidate may eliminate future bus transactions when a local update occurs, but both Write Update and Write Load must broadcast that a change has occurred whenever it occurs. Moreover, the Write Load can lower performance by displacing some item from a remote cache that would be more useful in the near future than the item just broadcast to the remote cache.

The ordering of efficiency is correct for many kinds of accesses, but it is undoubtedly in the wrong order for a barrier synchronization. Observe what happens when N processors attempt to synchronize at a barrier under the Write Invalidate protocol. As each processor obtains the barrier variable for updating, the processor places the barrier variable in its local cache. Shortly thereafter another processor reaches the barrier, requests the current value of the barrier variable, and invalidates the value in all other caches. Hence, the variable is invalidated in the cache where it had formerly just been updated, and the processor that just held that variable is forced to refetch it.

When the Nth processor attempts to update the barrier variable, it has to contend for the bus with $N - 2$ other processors that are trying to refetch the variable. In the worst possible case, the number of bus transactions can grow quadratically in N, but the growth reduces to only linear in N if all active requests for a variable are satisfied by one broadcast. Because each update of a variable invalidates all caches, at least N bus transactions have to take place to satisfy the barrier. The fact that a processor can determine when it has the exclusive copy of a variable is not important in this instance because each processor only updates a barrier variable once. The Write-Invalidate protocol is most useful when a processor updates a variable several times during one period of cache residence. Hence, the effectiveness of the Write-Invalidate protocol is wasted on barrier synchronizations. In fact the Write-Invalidate causes a problem because it forces all processors to rerequest the shared variable at the same time, and this tends to saturate the bus.

The second protocol, Write Update, is better in the sense that it does not remove an active variable from remote caches. In fact, it automatically delivers a new value for the remote processors to examine. When the barrier is finally satisfied, the update operation delivers the final value to active processors without requiring them to request the value individually. Hence, Write Update is distinctly better than Write Invalidate for implementing barrier synchronization.

The last protocol is actually slightly better than Write Update because it sends the current value of the barrier variable to remote caches prior to actual need. As each processor first comes to a barrier, it normally experiences a cache miss. If the first access is a read access, the Write Load protocol turns that access into a cache hit, and saves the cost of the miss and the cycle required on the bus. If the first access of a process is a decrement, then Write Load will be essentially the same as Write Update because under both protocols, the processor attempting to update the barrier variable must request a bus cycle to obtain write permission before updating the variable.

Because synchronizations, in general, require close cooperation among several processors, the Write Load and Write Update protocols will tend to yield better performance than Write Invalidate when used on synchronization variables. Eggers and Katz [1989] confirmed the findings in this discussion in a study that evaluated several different protocols by means of trace-driven simulations. They found that a protocol closely related to Write Load gave better hit ratios than Write Update, and attributed this to its use for synchronization purposes.

This short example illustrates that synchronization functions require close cooperation among many processors, and this can be sustained only with the right kind of information transferred at the right time. For multiprocessors with very few processors, it is possible to use a bus-based cache-coherence protocol for synchronization, but that protocol might not be the same one used for other shared variables. For example, one may choose to use Write Load for barrier variables and Write Invalidate for other kinds of variables. Because shared variables can be used in different ways, it is essential to match the cache-coherence protocol to the particular use of a shared variable.

For large-scale multiprocessing with 100 or 1000 processors, cache coherence may not be feasible to implement if it has to satisfy the needs of both synchronization and normal sharing. Consider the cache-coherence traffic in a 1000-processor system in which the interconnection network between processors has infinite bandwidth and no delay. If a broadcast invalidate or update is issued by a processor once every 100 clock cycles, then each processor receives 10 broadcasts per cycle on the average, and could receive up to 999 on any single cycle. Clearly, the broadcast has to be avoided in such a multiprocessor, yet the broadcast is the preferred mechanism for synchronization. The protocol has to be more selective and should not broadcast information to processors that do not need it. Even this type of protocol produces so many messages that the caches are kept busier by the cache-coherence protocol than by doing useful work in support of their local processor.

Because the bulk of the load on a cache-coherence mechanism may well be caused by synchronization operations, a practical way to build such a multiprocessor is to provide a low-cost subsystem dedicated to synchronization. If this successfully removes the bulk of the operations that otherwise would be performed by a cache-coherence network and protocol, then the operations that remain may be relatively easy to satisfy at reasonable cost. To satisfy a 1000-processor barrier by means of conventional cache-coherence techniques overburdens the cache-coherence network, and thereby severely degrades system performance. Yet a very simple low-cost dedicated network can implement the 1000-processor barrier with high efficiency. Clearly, specialized techniques for synchronization are quite attractive in highly parallel multiprocessors and they may open the way to practical implementations of cache-coherence protocols in such systems.

This completes the discussion of synchronization and cache coherence. The following sections return to techniques for writing efficient multiprocessor algorithms.

7.3 Parallel Search—How to Use and Not Use Parallelism

One of the most obvious ways to use parallel processors is for searching. Many researchers report excellent computation speeds in search applications, mainly based on the number of processors that are busy during the search process. Unfortunately, there is quite a difference between the number of processors busy and the true speedup in a multiprocessor since processors need not be doing useful work.

In this section we describe two different search algorithms. One is a search for a maximum of a function. For this problem it is rather surprising that the optimal search strategy yields only an O(log N) speedup. Even more surprising is the fact that all processors are busy during every step of the algorithm, so the magnitude of the wasted computing effort is not obvious. The second algorithm is a more sophisticated search algorithm. It is reproduced here to illustrate where one might look for useful parallelism.

7.3.1 Searching for the Maximum of a Unimodal Function

Karp and Miranker [1968] investigated the problem of finding the maximum of a unimodal function with N processors. A typical function to explore is shown in Fig. 7.7. By definition a unimodal function has a single mode or maximum located between its endpoints. Our objective is to find that maximum to within a unit interval on the x-axis. The search is to be conducted on a multiprocessor whose processors can evaluate $f(x)$ at any given x between the endpoints of the interval. We assume that the evaluation takes a fixed constant time so that all processors start and finish simultaneously. After evaluating the function, the processors can exchange information and determine the next point to evaluate. This too takes a fixed constant time.

The full search algorithm consists of a repetition of the processes that respectively evaluate and exchange information. The repetition continues until the maximum is pinned to within a unit interval. Karp and Miranker show that the optimum strategy depends on the parity of the number of processors, but whether that parity is odd or even, the optimum strategy produces an O(log *N*) speedup with respect to a single processor.

What is deceptive about this problem is that every processor is busy at every step, and we intuitively do not expect the final computation time to be so poor as to yield only an O(log *N*) improvement. In fact, with a sufficiently large number of processors we can pin the maximum to a unit interval in a single step—the ultimate in high speed. But since a single processor can find the maximum with a binary search in O(log *N*) steps, it becomes clear that O(log *N*) is all the speedup possible.

Figure 7.7 shows a typical situation during the execution of the algorithm. The vertical lines show where seven simultaneous probes are executed. The lines are uniformly spaced in this example. Karp and Miranker describe where the probes should be made for the optimum strategy, but the details of the optimum strategy are not important for this discussion.

What is important is the nature of the information returned. From the given

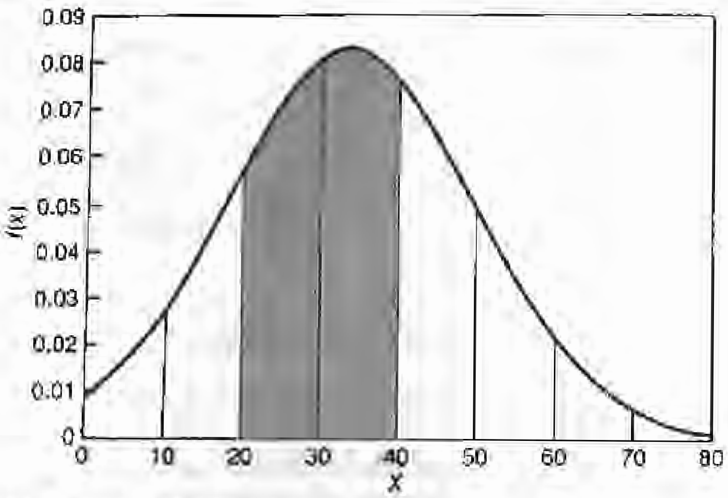

Fig. 7.7 Searching a unimodal function.

set of probes, we can conclude that the maximum must lie somewhere in the shaded region. The reason is that we can compute the derivative of the function by examining two neighboring values of the function. At the maximum of the function the derivative goes to zero. Only in the shaded regions can the derivative of a unimodal function become zero. Therefore, the next step is to assign the seven processors to evaluate the function in the shaded region and repeat the process.

A little reflection shows where the wasted effort is going. The only information actually used to guide the search is where the derivative changes sign. The outlying processors work as hard as the middle processors in evaluating the function, but the results produced by the outlying processors are of no value.

The only information extracted is the derivative of the function, and because the function is unimodal we know that if the derivative is negative at x, it is negative at all $y > x$. Consequently, if we find some x with a negative derivative, then the processors to the right of this point are wasting their effort. Similarly, if the derivative is positive at some x, it is positive at all $y < x$. Processors operating to the left of a point with a positive derivative are wasting their effort as well.

Let's examine the problem from the point of the view of the information available and the information actually used. Each of N processors returns essentially one bit of information, namely the sign of the derivative of the function. Thus in one step of the parallel algorithm we compute N bits of information. The bits, however, are not independent. In fact, the N processors create $N + 1$ intervals on the x-axis, producing exactly N possible choices for an adjacent pair of intervals to search on the next step. The amount of information in N choices is only log N bits, not N bits. Hence we expend the effort to produce N bits of information and obtain only log N useful bits. In essence, the algorithm throws away $N - \log N$ bits per iteration, which accounts for the wasted effort in this algorithm.

Is there a way to speed up this search? No, not if the constraints are obeyed. But there could be a way if other options are available. For example, the processors are constrained to evaluate $f(x)$. This is not satisfactory because it almost surely forces some evaluations to be useless. If the processors are given a different representation of $f(x)$ so that each evaluation gives independent information, the speedup might be greater. It might be possible, for example, for each processor to work with a Fourier transform of $f(x)$, which is helpful because each point in the transform contains information about all points of the function.

The fact that the function is unimodal forces the derivative information to be redundant. If the function were multimodal, and we had to find a global maximum, the work per processor would no longer be redundant because information produced about one region of the function sheds no light about the function in a different region.

The unimodal function is very important, however, because this is the function encountered in database searches for lookups by sorted key. When the search key is compared to a probe key, the difference is computed. The next point in the search depends on the sign of the difference. The absolute value of the difference function is unimodal, in this case having a single minimum instead of a single maximum.

Karp and Miranker's results show that multiple processors will not be very efficient if they are used to perform a search by making multiple probes to a file ordered by a single search key. Instead, multiple processors should be used to conduct independent searches. Therefore we cannot expect a multiprocessor to perform any single-key search much faster than a single processor can, but we can expect a multiprocessor to do many different searches in parallel with high efficiency.

Does the analysis suggest that multiprocessors are not useful for conducting parallel search for a single key? In some, but not all, cases, parallel search is indeed doomed to be inefficient and is reasonable for only small numbers of processors.

When a database is sorted by some key, the distance between the search key and a probe key is a unimodal function, so this problem definitely fits the Karp-Miranker model. When database keys are unsorted, we have the equivalent of a multimodal function, and the Karp-Miranker assumptions do not hold. A serial search might have to examine the entire database. In this case, a multiprocessor search has a potential for excellent speedup.

Therefore, we are tempted to take advantage of multiprocessors for search by using them on unsorted databases and claiming excellent performance. In this case, however, the savings from parallelism is not truly the speedup observed; it is the savings in the overhead used to sort the database and maintain that sorted order. If this overhead is small, then the effectiveness of the parallelism is small. If this overhead is large, then the parallelism is potentially beneficial. The actual choices available to the user thus are:

1. Use a serial computer, and use sorting or a database index to facilitate fast searching; or
2. Use a multiprocessor, and avoid the additional cost to sort the database or to produce an index.

We must compare the cost and performance of these two alternatives in order to evaluate parallel searching. We should not compare parallel search to ordinary serial search unless serial search is truly the only other alternative.

In many business applications the cost of sorting or building an index can be amortized over hundreds or thousands of searches. Rarely in such instances does it pay to perform parallel search. On the other hand, some problems in cryptography are essentially enormous searches that are only performed once per database. The equivalent of building an index (or sorting the database) is far more costly than searching the database in parallel by using a multiprocessor.

A comparison of parallel and serial search in the recent literature was stimulated by an article by Stanfill and Kahle [1986] regarding a highly parallel search of a large data base. The solution proposed by Stanfill and Kahle involved the use of a Connection Machine with 64,000 1-bit processors to search a multigigabyte database, but their solution required the entire database to be read from disk while doing the search. Because disk operations are very slow compared to operations done at the clock speed of a processor, the enormous cost of reading an entire data base from disk almost certainly cannot be offset by the speed gain due to parallel search. Boral and DeWitt [1983] brought this fact to light, and questioned the practicality of a parallel database machine until technology can provide a much faster auxiliary storage.

Nevertheless, Stanfill and Kahle implemented a parallel search of the type that Boral and DeWitt indicated would not be efficient. The weakness of the Stanfill-Kahle approach was observed independently by Stone [1987] and Salton and Buckley [1988]. Stone's analysis suggested that just one of the 64,000 processors working with the same total memory of the Connection Machine could perform the same task somewhat faster if it used an index in order to reduce the total traffic from disk. Salton and Buckley's analysis demonstrated that a low-cost workstation had roughly comparable performance as the Connection Machine when the workstation used an index. In both studies the gain in performance is strictly due to the much smaller volume of data actually read from disk. An algorithm that succeeds in keeping 64,000 processors busy is not necessarily a fast algorithm—the processors have to be performing useful work.

The important observation here is that parallelism is only one of many possible techniques for solving a problem. It may fare badly with respect to good serial techniques. Performance evaluation is crucial in judging the effectiveness of parallel programs. The comparison must always be done by seeking good serial algorithms against which to compare the parallel algorithms.

The ultimate quality measure of a parallel algorithm is performance per unit cost, not just performance alone. All algorithms for all processors can be reduced to this common measure. While it may be interesting to learn that a 1024-processor search is faster than a serial search, it becomes far less interesting when we discover that the speedup over a serial search is a factor 10, and that we can obtain a factor of 5 speedup by using only 32 processors.

7.3.2 Parallel Branch-and-Bound—The Traveling-Salesman Problem

A remarkable algorithm for solving the Traveling-Salesman Problem provides an excellent example of where and where not to exploit parallelism. The Traveling-Salesman Problem is rather deceptive because it is easily described and simple in concept, but extremely difficult to solve. The problem is to find a minimum-distance tour of N cities that visits each city exactly once and returns to the first city on the tour at the end. The problem input is a list of the distances between each pair of cities.

It is well known that this problem belongs to the class of hard problems known as *NP-complete,* for which the best available algorithms exhibit a worst-case computation time that grows faster than any polynomial function of the size of the input [*see* Aho, Hopcroft, and Ullman 1974]. Many researchers believe that the computation time for NP-complete problems actually grows at least exponentially with the size of the input, but this question is unanswered at this writing.

The algorithm we describe is remarkable because its average complexity is only $O(N^3 \log N)$ on a class of randomly selected input problems which is less than quadratic in the size of the problem since the problem size is $O(N^2)$. This appears to contradict the findings that the problem is NP-hard, but there is no contradiction.

The algorithm has a low complexity on the average, but its worst case may require exponential time, even though this event is extremely unlikely. The algorithm is from D. R. Smith [1984], who proved the results on average time and demonstrated that these results are consistent with actual running times on randomly generated sets of problems. The analysis might not hold for a class of problem instances whose characteristics are rather skewed and are not adequately represented by the more uniform distributions assumed in Smith's analysis.

The branch-and-bound technique executed on a serial processor is illustrated in Fig. 7.8. The algorithm depends on a subroutine that can compute the least-cost permutation for visiting N cities. We use the notation (1 2 3) to describe a route that visits City 1, then City 2, then City 3, and then returns to City 1. We call such a visit a *cycle* because its starting point is the same as its finishing point. We call a *permutation* of the cities to be a set of cycles such as (1 2 3)(4 5 6 7), such that every city appears in exactly one cycle.

A permutation is not necessarily a tour because in this case if you start at City 1, you return to City 1 after visiting Cities 2 and 3, and without having visited any of the other cities. A tour has to visit all of the cities exactly once. Obviously, a tour is a permutation that has but a single cycle, such as the permutation (1 2 3 4 5 6 7) for seven cities.

The subroutine that finds the least-cost permutation finds a permutation whose sum of city-to-city distances is the minimum among all permutations of the cities. The reason for finding the least-cost permutation is that it gives a lower bound on the shortest tour. Since a tour is a special kind of permutation, the shortest tour for a given problem cannot be shorter than the least-cost permutation.

Finding the shortest tour is extremely difficult, but finding the least-cost permutation is relatively easy [*see* Lawler 1976]. This takes only $O(N^3)$ time the first time we execute the subroutine. On subsequent executions, the input data will be only marginally different. Only $O(N^2)$ additional work is required to obtain the solutions for these subroutine calls. Lawler shows that the discovery

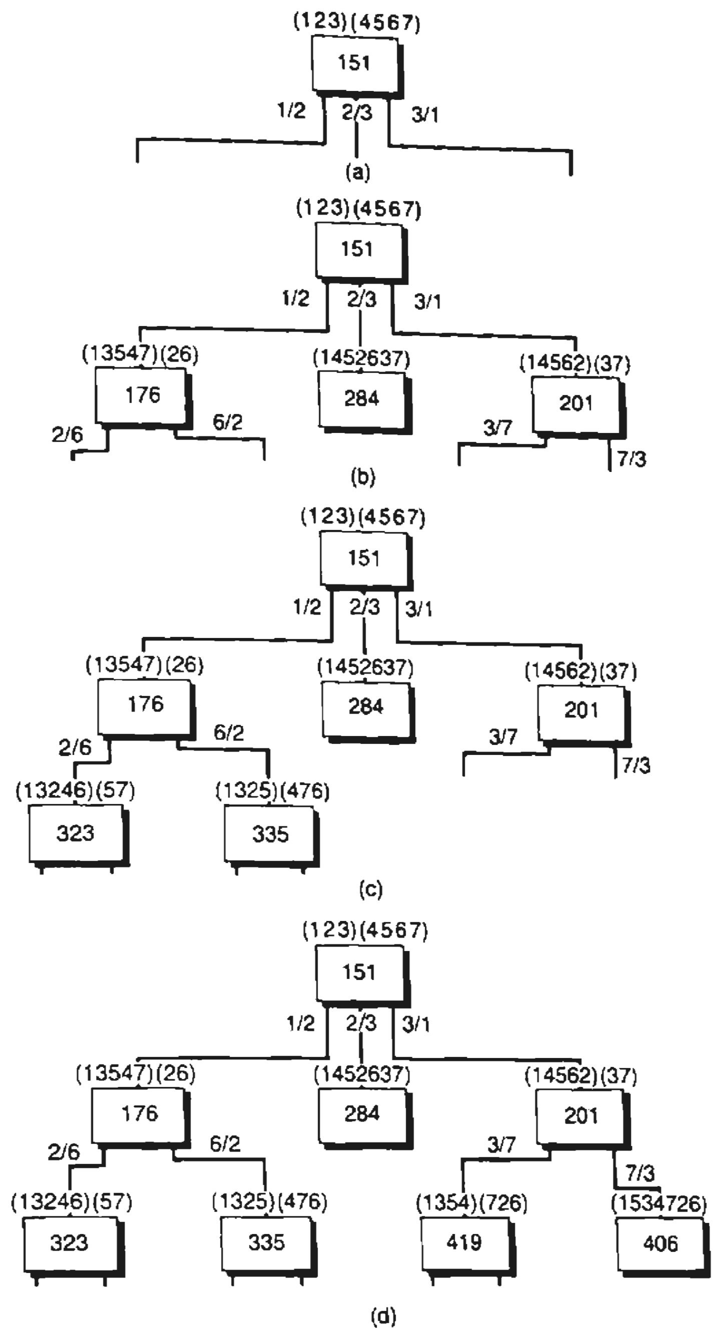

Fig. 7.8 Branch-and-bound search for the Traveling-Salesman Problem:
(a) Initial solution of the problem (three subproblems open);
(b) After examining the three subproblems;
(c) After expanding the two leftmost solutions; and
(d) The search after expanding the node for permutation (1 4 5 6 2)(3 7).

of the least-cost permutation, which in his terminology is the *assignment problem*, reduces to a minimum-cost, network-flow problem [Ford and Fulkerson 1956], which is solved by repeated applications of Dijkstra's shortest-path algorithm [1959].

Figure 7.8 illustrates how the lower bound information is used. In Fig. 7.8(a), we show a single node of the search tree labeled by the permutation (1 2 3) (4 5 6 7), with a total distance of 151 shown inside the node. The algorithm produces this number by running a least-cost permutation algorithm on the original algorithm. (To prevent solutions with one-city cycles, the original problem has an infinite cost of going from any city to itself.) Since (1 2 3)(4 5 6 7) is not a tour, the best tour has an equal or higher cost. The least-cost tour must differ from this permutation on at least one branch of each cycle, so without loss of generality, we examine the shortest cycle, which in this case is (1 2 3).

The least-cost tour differs from this cycle in at least one way, and possibly in more ways. That is, either the tour does not go from City 1 to City 2, from City 2 to City 3, or from City 3 to City 1. These three possibilities are shown in Fig. 7.8(a) as three labeled arcs leaving the original node.

Since at least one of these three roads is not on the least-cost tour, we can create three new subproblems to investigate. In each of three subproblems we eliminate the possibility that one of the three roads of interest is in the least-cost permutation. Figure 7.8(b) shows the result of this step.

The leftmost node at the second level shows what happens when the distance from City 1 to City 2 is made infinite. When we call the least-cost permutation subroutine with this new condition, it reports back that the least-cost permutation is (1 3 5 4 7)(2 6), with a cost of 176. Note that the road from City 1 to City 2 is not on this permutation because that road happens to be infinitely long.

When the road from City 2 to City 3 is infinite, the least-cost permutation turns out to be a tour with a cost of 284. Although a tour has been produced by the algorithm, the tour is not necessarily the least-cost tour for the original problem. Additional work is required to show that this tour is optimal or to find a lower-cost tour.

When the road from City 3 to City 1 is infinite, the least-cost permutation is (1 4 5 6 2)(3 7), with a cost of 201. Although we have now discovered a tour that has a low cost, it might not be the least-cost tour. Both of the other subproblems are open to the possibility that further exploration of these candidates could yield tours of cost lower than 284, although we know now that no tour can have a cost lower than 176.

To investigate the leftmost node, note that the permutation can be broken at its shortest cycle by opening the road from City 2 to City 6 or by opening the road from City 6 to City 2. (These roads do not have to be the same road.)

Similarly, the rightmost node can break the cycle that contains City 3 and City 7 by opening the road either from City 3 to City 7 or City 7 to City 3. Thus there are four search paths that warrant further exploration. The two best candidates are the descendants of the leftmost node in the figure because this node has the least bound of any node on the perimeter of the search tree.

Figures 7.8(c) and (d) show what happens when we follow the four open subproblems. Searching beneath the node with the lowest bound obtains two new permutations, whose costs are 323 and 335, respectively. Note, for example, that the leftmost permutation is (1 3 2 4 6)(5 7), which is the least-cost permutation for the case in which City 2 does not follow City 1 and City 6 does not follow City 2.

At this point, the rightmost node has the lowest bound, and the search branches to that node for further exploration. Examining the two subproblems of this node produces two new permutations, whose costs are 419 and 406. The latter permutation happens to be a tour. None of the subproblems yields either a tour or permutation whose cost is lower than the cost 284 for the least-cost tour discovered in Fig. 7.8(b). Hence, the tour (1 4 5 2 6 3 7) is optimal and the problem has been solved.

Although this highly contrived example is not necessarily typical of real problems, the power of the branch-and-bound algorithm is quite clear. By expending $O(N^2)$ time at a node, we can find out how expensive a tour might be if we examined the descendants of that node in the search for a tour. If a bound is very high, the search path is not promising, and we can abandon the search from that node.

In Fig. 7.8, there are $6! = 720$ distinct tours of the 7 cities, and the bounding operation eliminates 718 of them from consideration. We do not claim that the algorithm behaves this efficiently in general. But D. R. Smith [1984] does claim that the average number of times that a least-cost permutation is generated is $O(N \log N)$ although the proof is not in this article. With a cost of $O(N^2)$ time to generate a least-cost permutation, the total time for the algorithm on the average is $O(N^3 \log N)$.

Since Smith's results assume an unbiased distribution of problems, his results may not hold for problem distributions with strong statistical biases. Nevertheless, let us assume that Smith's results hold for a particular set of problems and consider how parallelism can be put to effective use.

The search tree in Fig. 7.8 in general has $(N - 1)!$ leaf nodes, one for each possible tour, assuming that each tour starts at City 1. Therefore its depth is $O(N)$ if the average branching factor is proportional to N, and its depth is $O[\log (N!)] = O(N \log N)$ if the average branching factor is not larger than a constant that does not depend on N. If the depth is $O(N)$, then we can say that on the average we examine $O(\log N)$ parallel paths while visiting $O(N \log N)$ nodes. This suggests that as many as $O(\log N)$ paths can be usefully examined concurrently in a multiprocessor. If we expend $O(N)$ processors to examine the open subproblems, we would obtain a useful speedup of only $O(\log N)$, and the speedup is similar to the Karp-Miranker problem.

Note in Fig. 7.8(b) that we can commit two processors simultaneously to the open subproblems for the leftmost node. We can also commit two processors to the rightmost node. However, if the leftmost node returns tours whose cost is lower than 201, which is the cost of the rightmost node, then any additional computation expended on the rightmost node is wasted effort. This situation is analogous to the wasted effort in the Karp-Miranker search.

If the depth of the search tree turns out to be of $O(N \log N)$, then the possibility of using parallelism effectively to explore multiple paths is rather unpromising. On the average only one path is actively pursued by a serial search in this case, and if multiple paths are pursued concurrently, all path computations but one are almost surely wasted.

The obvious way to apply parallelism is to apply all processors to the computation at one node to perform the evaluation of the least-cost permutation. An efficient approach is to examine only the nodes that a purely serial algorithm examines. This ensures that no effort is wasted examining other nodes.

In the process of examining a node, apply as many processors as can be applied efficiently to find the least-cost permutation. That number may vary with the architecture, depending on communications and access to shared variables.

Dijkstra's shortest-path algorithm can be executed with a speedup of $O(N/\log N)$ on some N-processor parallel architectures, assuming that contention for shared resources does not produce excessive performance degradation. The speedup, however, is architecture dependent. If an architecture can produce a speedup of $O(N/\log N)$ or better for Dijkstra's shortest-path algorithm, then this architecture will produce a very fast, efficient parallel solution of the Traveling-Salesman Problem, provided that the statistical distributions of the problems to be solved are similar to those assumed by Smith.

In this example, the key observation is that searches along parallel paths are not independent and can produce wasted effort, whereas there is an opportunity for parallelism in performing the work along one path. Pick a promising candidate and focus the computing power on this candidate, rather than spread the computation across several candidates.

7.3.3 Speedup and Parallel Complexity

We have stressed efficiency in parallel computation, and have used speedup as a means to express efficiency. While it is an excellent single measure, speedup measures when used improperly can be misleading.

For example, consider an FFT algorithm whose serial complexity is $O(N \log N)$ for N data points. Using a technique such as described by Pease [1968] we can construct an N-processor computer that computes an N-point FFT in a time proportional to $\log N$. Thus, the parallel computer achieves a speedup of N, and the efficiency is excellent.

Now consider a different problem and perform a similar analysis. Let this problem be Problem X for which there exists some very efficient serial algorithm that solves any instance of Problem X of size N in time $O(N^3)$ in the worst case. We say that the complexity of the algorithm is $O(N^3)$. Assume that we are very fortunate, and are able to demonstrate that no algorithm exists that has a lower complexity. Consequently, we have a serial algorithm against which we can compare parallel algorithms.

After some careful study, assume that we produce a novel parallel computer and a suitable algorithm for that machine that work extremely well together to solve Problem X. The match is so good that an N-processor version of the computer system can solve any instance of Problem X of size N in a time $O(N^2)$ in the worse-case. In other words, the parallel complexity of the algorithm on the computer system is $O(N^2)$. The research community quickly endorses this as an efficient scheme, and heralds it as having an $O(N)$ speedup.

But some surprises lie ahead. We build a 100-processor version of the machine, implement the algorithm, and confirm that it is working correctly. Then we enter real data of size 100. We run the parallel algorithm on the 100-processor machine and run the same problem on a serial machine that implements the efficient serial algorithm. The serial and parallel machine use the same level of device technology. Our expectation is that we achieve a large speed-up, not necessarily a 100-to-1 because we have not accounted for constant factors, but nevertheless we expect to see the parallel algorithm running much faster than the serial algorithm. But we do not achieve any speedup at all. The parallel implementation seems to run somewhat slower. Where is the N-fold speedup? Perhaps the constant factors are working against us in this instance.

We explore further. We run many problems and we let the problems grow and shrink in size. We also vary the number of processors. As the performance picture becomes clearer we discover that the speedup is not $O(N)$ but only $O(\log N)$. How can this be true?

We have been misled because the problems that we have been using to test the algorithm are not the same problems that determine the complexity of either the serial or parallel algorithms. When we say that an algorithm has a complexity of $O(N^3)$, we are only saying that the most difficult possible input configuration of size N can be solved in a time that grows as N^3. We have presented no information on other input configurations. These may or may not be as difficult to solve. Similarly, to say that the parallel complexity is $O(N^2)$ is the same as saying that the most difficult problem to solve in parallel can be solved in a time proportional to $O(N^2)$. The most difficult problem to solve in serial need not even be related to the most difficult problem to solve in parallel.

When we attempted to test the parallel program, we happened to select problems in a way that produces an average serial complexity of $O(N^2 \log N)$. For this selection of problems, the parallel algorithm happens to produce only $O(\log N)$ speedup, and runs in a time proportional to $O(N^2)$, which does not violate the claim that the parallel complexity is O(N^2). The constant coefficients are just different enough to cause the first parallel test to run slower than the serial algorithm, and the full battery of tests confirmed that true speedup is O(log N).

Although this example sounds hypothetical, it is illustrative of actual research results reported in the literature. In the case of parallel algorithms for the FFT, the speedup measure is accurate because all input configurations run in the same amount of time. That is, the worst case, the best case, and the average case are all the same.

In the case of Problem X, the worst case and the average case are far different. A measure of complexity based on the worst case misleads us when we produce instances of the average case. Even if we had produced instances of the worst-case parallel input, we may still be in trouble because the worst-case parallel input is not necessarily the worst-case serial input. Hence, the worst-case parallel input need not produce the speedup of O(N). The only sure way to see N-fold speedup is to select instances of the worst-case serial input. Though we are promised N-fold speedup for these inputs, the speedup for all other inputs is totally unknown from the information at hand. The algorithm we treated originally as a breakthrough may be only a useless curiosity, or alternatively may eventually be shown to meet our original expectations. We cannot tell until we investigate the behavior of the parallel algorithm on a realistic set of data inputs.

The important lesson is to understand the limitations of the speedup measure. Does the speedup reported for a parallel algorithm hold for all input data, typical input data, or for some small subset of input data? Unless the measure holds for all input data, the measure gives an incomplete picture of the efficiency of an algorithm. To be a useful measure, the measure should report on typical input data, whereas the tendency in the literature is to report on worst-case input data. We must be able to define and analyze the typical case, but rarely can we find the characteristics of the typical case. This is the difficulty that has led to worst-case studies, and in turn has produced speedup measures whose true significance is still unknown.

7.4 Transforming Serial Algorithms into Parallel Algorithms

In putting multiprocessors to use, a major hurdle is writing programs for such architectures. In the worst case, every problem has to be studied anew and solved by an algorithm implementation tuned to a particular architecture. This technique will certainly be used for the very largest problems, which consume days or weeks of computation time, because the human effort expended to optimize the algorithm is paid back by a large reduction in computer time. But for more moderate problems, those that take a fraction of an hour, for example, the human effort to optimize the algorithm might save only a few minutes of computation, which may not be worthwhile. Therefore, a major objective is to use programmed transformations to produce reasonably good parallel programs from serial programs.

One way to automate the production of parallel programs is to construct a compiler for a standard high-level language to produce output for a multiprocessor. With such a compiler, existing software libraries can be mapped to a multiprocessor with a minimum of effort. Some fraction of the library undoubtedly will exhibit negligible parallelism and will produce rather inefficient parallel implementations. These programs can be run serially.

The interesting programs are those that yield efficient parallel codes. The codes need not be as efficient as hand-coded versions of the programs, provided they come within a factor of 2 to 5 of a hand-coded translation. If the inefficiency is as high as a factor of 10, the compiler is still useful as a stopgap tool that provides a fast way of producing programs for a parallel architecture. The inefficient translations it produces eventually have to be reprogrammed by hand or by a better compiler to create versions that are satisfactory for production use.

Creating a high-quality optimizing compiler for a multiprocessor is a formidable task. An early attempt by Kuck *et al.* [1972] showed that there is easily exploitable parallelism on the order of 10 to 100 in many ordinary FORTRAN programs. The next decade produced far more sophisticated developments that have been used extensively for real applications.

For vector architectures, leading work by Miura [1986], a student of Kuck's, for Fujitsu vector processors and by F. Allen for the IBM 3090 vector processor produce code that is nearly as efficient as the best programmers can produce and is much more efficient than can be produced by inexperienced programmers.

Compilers for multiprocessors have lagged behind compilers for vector processors because the translation problem is far more complex for multiprocessors. Vector compilers find a way to do one operation simultaneously across many processors; multiprocessor compilers find a way to do many operations across many processors at unpredictable times. The thread common to the two types of compilers is that they need to identify dependences from statement to statement to determine the order in which events can be scheduled.

For vectorizing compilers, published work by Kuck *et al.* [1984], J. R. Allen *et al.* [1983], J. R. Allen [1983], and Padua and Wolfe [1986] illustrate the underlying theory and the directions taken by compiler writers. The actual art of vectorizing compilers is more advanced than the literature indicates, but the literature captures the most important and useful transformations. Cytron [1984] and Padua and Wolfe [1986] address the problem of optimizing code for multiprocessors.

7.4.1 Dependence Analysis

The most fruitful way to obtain parallelism in serial programs is by executing loop iterations across several processors. We illustrate this technique earlier in this chapter, and we also introduce the notion of *chunksize* in Program 7.4 to show that one processor can execute a group of iterations instead of a single iteration. Although other forms of parallelism exist and are potentially detectable by an optimizing compiler, in typical applications the bulk of the speedup obtained from parallelism is through the parallel execution of loop iterations. Therefore, we focus on ways to perform loop iterations in parallel in this text.

An optimizing multiprocessor compiler has the task of detecting parallelism, but the task name is misleading. The compiler actually detects serial behavior, and, by default, everything left is potentially executable in parallel. To produce parallel code for a loop iteration, the compiler has to detect when successive iterations have to be executed serially. As an example of dependence analysis, consider the loop:

```
For i := 1 to N do
  begin
    A[i] := A[i − 1] /B[i];
  end; {* do loop *}
```

As written, each iteration depends directly on the prior iteration because a variable written in the prior iteration is read by this iteration. This is WRITE/READ dependence. Other dependences possible are READ/WRITE and WRITE/WRITE. The READ/WRITE dependence requires the variable to be read by a prior iteration before it is written by this one, and the WRITE/WRITE dependence forces the value of the variable to be written last by the present iteration rather than by a prior iteration.

The dependence in the example is very easy for a compiler to detect because it is forced by a single variable. Other examples lead to more complex cases, such as an iteration with the following statement:

```
A[i] := A[C[i]];
```

In this case, the dependence is READ/WRITE if C[*i*] is less than *i* and WRITE/READ if C[*i*] is greater than *i*. Moreover, if the values in C are computed during execution, the compiler cannot determine which dependence exists and therefore cannot optimize the code. Therefore, the compiler can detect loop-to-loop dependences only when all subscript expressions in an iteration and the loop increment have values known to the compiler. Optimizing compilers are forced to assume that the dependences are present if index variables depend on execution-time program behavior. Otherwise, the optimization process is likely to produce a translated program that runs incorrectly.

A general procedure for detecting dependences is to list the names of the variables read and written in a loop iteration. If a name appears on both lists, it potentially leads to a READ/WRITE or WRITE/READ dependence. All variables that are written are potentially WRITE/WRITE dependences. The compiler has to examine each case further to determine if an actual dependence exists.

For the WRITE/WRITE dependence to exist, one variable has to be written by two different loop iterations. This situation usually has two distinct statements in the loop, such as

```
A[i] := B[i]/10;
A[i - 1] := C[i] + B[i];
```

With both statements present in one iteration, it becomes clear that the prior iteration, using the value $i - 1$ as an index value, writes $A[i - 1]$ and $A[i - 2]$, leading to the WRITE/WRITE dependence for variable $A[i - 1]$. Note that we assume that the loop index is increased by 1 during each iteration. If the loop index is increased by 2, then there is no dependence caused by writing two successive values into A. READ/WRITE and WRITE/READ dependences are equally easy to detect as WRITE/READ dependences.

7.4.2 Exploiting Parallelism Across Iterations

In this section we show how to use dependence information to guide the translation of serial programs into multiprocessor programs. There are just a few techniques given here, but they are widely useful and produce the bulk of the speedup obtainable in typical programs. However, there are many other techniques not discussed in this section that are also of value, especially techniques designed for specific classes of programs. Interested readers will find J. R. Allen [1983], Cytron [1984], and Padua and Wolfe [1986] useful in-depth treatments of the topic.

Our objective for a multiprocessor is to split apart iterations that are independent. This boosts speedup, provided that independent iterations have a sufficiently high *R/C* ratio. We also want to chunk iterations together into larger tasks to boost efficiency by improving the *R/C* ratio when this also boosts performance, even if it reduces parallelism. The ideal situation is to chunk dependent iterations together into large tasks in a way that creates a collection of independent large tasks.

As an example of this idea, consider Program 7.10. The program is shown as it would probably be found in a program for a conventional serial machine. We assume that the program uses neither **do seq** nor **do par** phrases, described earlier in this chapter, because it is written specifically for serial execution.

A straightforward dependence analysis shows that Column 0 of matrix A is the cause of the dependences. There is a WRITE/READ dependence from iteration (i, j) to iteration $(i, j+1)$ because $A[i, 0]$ is both read and written for these iterations. This suggests that successive iterations in the serial program have to be executed serially.

Program 7.10 Computing row sums of a matrix.

```
for i := 1 to N do
  begin
    A[i,0] := 0.0;
    for j := 1 to N do
      A[i,0] := A[i,0]+A[i,j];
  end; {* i loop *}
```

Notes:

1. Matrix A is $N \times N$, with indices running from 1 to N.
2. The sum of Row i is computed and stored in $A[i,0]$.

A sophisticated compiler should detect that there are no dependences due to the i index, so the i and j loops can be interchanged, as shown in Program 7.11. The inner loop satisfies the READ/WRITE dependence on the index i. To ensure that Column 0 is properly initialized, it is initialized separately in an earlier loop. Note that successive serial executions of the inner loop can be chunked together into a single task that does all N iterations for one value of i. Each of these large tasks is independent and can be executed concurrently.

It is also possible to obtain greater parallelism by observing that the inner loop can be chunked into several medium-size tasks, for example, k of them, so that each form the sum of N/k row elements. For a particular value of i, the variable $A[i, 0]$ is a variable shared across k tasks, which forces serialization of the tasks because of a READ/WRITE conflict.

A clever compiler can detect that the summation into the row sum can be done in any order and can change strictly serial execution of the k tasks into parallel execution, with each task computing a local sum that is added to the shared variable at the end of the chunk. The addition at the end is controlled by a LOCK/UNLOCK, Compare-and-Swap, Fetch-and-Add, or other similar means. The value of k should be selected to reflect the available parallelism and the best choice for the R/C ratio.

The key idea illustrated by this example is to observe the essential dependences exhibited by the algorithm. The order of execution is free to be changed, provided that the dependences are satisfied. In the example, the order of indexing of the loops is changed, which is a common situation among algorithms. By changing the order, the transformed program structure has N parallel tasks (or kN if chunking is used), instead of N^2 serial iterations. Not only is the transformed program more parallel, but its R/C ratio can be adjusted to minimize synchronization inefficiency.

Program 7.11 Computing row sums of a matrix, transformed version.

```
for i := 1 to N do
  A[i,0] := 0.0;
for j := 1 to N do
  begin
    for i := 1 to N do
      A[i,0] := A[i,0]+A[i,j];
  end; {* j loop *}
```

Notes:

1. Matrix A is $N \times N$, with indices running from 1 to N.
2. The sum of Row i is computed and stored in $A[i,0]$.
3. For a multiprocessor, the loop on i can be chunked together to make larger tasks which improves *R/C*.

As a second example, let us return to the familiar example of the inner loop of a Poisson solver. In Program 7.1, the item updated depends on its north, east, south, and west neighbor. No matter how we choose the iterations, by row or by column, ascending or descending, we will have READ/WRITE and WRITE/READ conflicts. Therefore interchanging the iterations is not particularly helpful for this program.

There is, however, a parallel structure that can be exploited here. If the cells of the matrix are laid out on a checkerboard, then the iteration in Program 7.1 shows how to update a black square by averaging the values in its neighboring red squares, and similarly, how to update a red square by updating the values in its neighboring black squares. The red and black squares form two independent sets of variables, since no red square depends directly on a red square, no black square depends directly on a black square.

Therefore, a possible approach is to create a task that updates red squares from black ones, and another task that updates black squares from red ones. The two tasks can be divided into smaller tasks by chunking indices, and the chunksize should be chosen to reflect available parallelism and *R/C*. The iteration of Program 7.1 can be done by updating the black squares, then updating the red squares, with each update done across the available processors. Barrier synchronization is required at the end of an update of each color.

The parallel computation using red and black squares produces an iteration that is not quite identical to the iteration given in Program 7.1. Note that as each point is updated, two of its four neighbors have already been updated. For example, at Row i, Row $i - 1$ has new data already, but Row $i + 1$ has not been updated. So the north and west neighbors of each point are new, and the south and east neighbors are old. This iteration is called the *Gauss-Seidel* iteration [*see* Varga 1962].

Another possible technique is to compute the updated data for the entire matrix before making any update. Such a scheme, called the *Jacobi* iteration, uses old data for all neighbors. The red-black scheme is equivalent to selecting new data for all neighbors. In typical situations, all three schemes converge to the same solution at different rates. The red-black scheme converges the fastest because it uses new data more quickly than do the other two schemes. The slowest convergence occurs for the Jacobi iteration.

In general, iterative calculations such as the one illustrated in Program 7.1 may converge or diverge, or they may oscillate while neither converging nor diverging. If the numerical conditions are such that convergence occurs, then in general, the more new data used in an iteration, the faster the convergence will be. Thus, the transformation of Program 7.1 to one that uses a red-black ordering and executes in parallel on a multiprocessor is likely to be an effective transformation. If this is done automatically, the program should produce a warning that the iterative method has been altered in the transformation.

The red-black scheme for Program 7.1 is ideal for multiprocessor use. Because half of the points in a mesh can be executed in parallel, the program can be split across any reasonable number of processors, and the chunksize can be set large enough to keep synchronization overhead small. In a multiprocessor, irregular boundaries and special regions within the mesh are treated easily and far more efficiently than in a vector architecture that broadcasts one instruction to all processors.

Is it reasonable to assume that an optimizing compiler is clever enough to change an iteration from one form to another? If the optimizing compiler is used for general-purpose computation, the answer is no. There are literally hundreds of useful transformations that could be applied, which is far too many to incorporate in a compiler.

However, if the compiler is dedicated to a specific class of computations, such as partial differential equations, it is quite reasonable to incorporate within the compiler the most useful transformations that occur in practical problems. In this case, the transformation of the Gauss-Seidel iteration in Program 7.1 to a red-black iteration is frequently done by hand, and it should be known to the compiler writer.

Optimizing compilers may be viewed as programs that have a repertoire of tricks to apply, and they do their work by searching through their bag of tricks for the most appropriate ones to apply. A clever researcher might discover a new trick, such as the red-black transformation, which no compiler can discover on its own. Once the trick is known and published widely, the compiler writer can add the new trick to the compiler's repertoire. The compiler might not be very good when it is first completed, but as the bag of tricks grows, the compiler may be able to produce better parallel code than can most programmers.

Nevertheless, we may insist that any program transformation must leave the iteration unchanged. If this is the case for Program 7.1, we need a parallel program that does the Gauss-Seidel iteration. Lamport [1974] observes that a diagonal scheme, as shown in Fig. 7.9, is equivalent to the Gauss-Seidel iteration. In Lamport's scheme, the matrix is not scanned by rows or columns, but by diagonals. Along any diagonal, all the points depend on the previous and next diagonals. The previous diagonal holds the north and west neighbors; the next diagonal holds the south and east neighbors. Since each diagonal sees new data from the prior diagonal and old data on the next diagonal, the iteration that marches from diagonal to diagonal is a Gauss-Seidel iteration. Lamport shows that the transformation of a program written in the form of Program 7.1 into a diagonal scan can be incorporated into a compiler and fully automated.

The diagonal scheme has a serious disadvantage because some diagonals are very short and severely limit parallel execution. Recall that there has to be Barrier synchronization along a diagonal to ensure that one diagonal is completely updated before the next diagonal is started. Lamport, however, shows that it is possible to combine two diagonals N apart, to obtain a total of N points to update, lying on two different diagonals that can be updated simultaneously. In the first pass across the diagonals, this algorithm updates Diagonals 1 through N, one at a time. When Diagonal $N + 1$ (of length $N - 1$) is reached, it is paired with the second iteration of Diagonal 1, to produce work for N points. Next,

1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	1
3	4	5	6	7	8	1	2
4	5	6	7	8	1	2	3
5	6	7	8	1	2	3	4
6	7	8	1	2	3	4	5
7	8	1	2	3	4	5	6
8	1	2	3	4	5	6	7

Fig. 7.9 Lamport's diagonal sweep for the Poisson problem on a square. The number within each cell identifies the iteration in which the cell is updated. This algorithm is equivalent to the Gauss-Seidel iteration because the north and west neighbors have new data, and the south and east neighbors have old data. By scanning two diagonals concurrently, the number of data treated in each operation is constant.

Diagonal $N + 2$ is paired with Diagonal 2 to produce another set of N points for updating. This continues through the last iteration, during which it is not necessary to update the first N diagonals. Although all N points along the two diagonals can be updated independently on N processors, they can be chunked together arbitrarily to match the parallelism to the architecture and raise the R/C ratio, if necessary. If the number of processors available exceeds N, it is possible to update odd-numbered and then even-numbered diagonals in parallel and obtain greater use of parallelism.

7.4.3 The Effects of Scheduling on Parallelism

The last topic we consider in this section is from Cytron [1984], who considered the effects of scheduling on parallelism. The idea is to schedule dependent tasks so that dependences are satisfied, and yet tasks are executed at least partially in parallel.

As an example of the use of scheduling, consider any loop body in which there is a WRITE/READ dependence from one iteration to a later iteration. A typical loop of this type has statements of the form

```
A[i] := B[i - 1]
B[i] := C[i];
```

In this example, Iteration i cannot begin until the prior iteration has written the value of $B[i - 1]$. If these two statements form the entire iteration, then Iteration i cannot start until Iteration $i - 1$ has ended. This is how we expect iterations to execute when dependences are discovered. But Cytron points out that lengthy iterations can be partially overlapped.

In our example, the 2 statements could be the first of 20 statements, rather than the only statements in the iteration. If so, and if no other dependences exist from iteration to iteration, then Iteration i can begin while Iteration $i - 1$ is executing, provided that Iteration i waits until $B[i - 1]$ has been computed.

The overlapping of iterations is analogous to pipelined execution of vector operations, except that the operations within one iteration can be arbitrarily complex, and the delay between initiations of successive iterations has to be long enough to satisfy the dependence constraint.

A compiler that exploits this form of parallelism has to be able to control execution-time scheduling in some way. In the compiled code it can produce an interrupt, message, or other form of control information at the point that a dependence is satisfied. The control information should be transmitted to a scheduler or equivalent task to force the release of a task waiting for the update to complete. The added overhead of the control information has to be low enough to make concurrent execution worthwhile. There is no point in seeking concurrent operation if the control required is extensive enough to create its own bottleneck.

7.5 Final Comments on Multiprocessors

This brings us to the close of this chapter. We have only presented a small portion of the current state of multiprocessor architecture, but we believe that the highlights discussed in this chapter give an accurate picture of the potential and pitfalls of multiprocessors.

Problems of overhead and effective parallelism are serious problems, and they are likely to limit multiprocessors to relatively few processors in practical systems. The 1000-processor system can become a reality in years to come, but much research is necessary in the interim to solve problems related to efficiency. Exploitation of multiprocessors depends strongly on finding ways to:

- Eliminate the MSYPS bottleneck;
- Reduce overhead for scheduling tasks;
- Solve the cache-coherence problem or to find an alternate means of providing fast local memory;
- Map serial programs to parallel programs; and
- Identify useful parallelism, as opposed to parallelism that leads to wasted effort.

As progress is made on these fronts, the multiprocessor becomes more attractive and eventually could be the architecture of choice for high-performance systems.

In earlier chapters we discuss six technology constraints that have to be overcome in an architecture. Some of the constraints are included in the problems preceding list. Overall, the comparison is as follows:

1. *Processor bandwidth:* processor bandwidth is extremely satisfactory for the multiprocessor because each distinct processor in the architecture has the potential to supply the full processor bandwidth to a problem. This facet of the architecture is one of its strengths.
2. *Memory bandwidth:* available memory bandwidth depends strongly on the mechanism for multiple accesses to memory. If no memory is shared, gross bandwidth is very high since it is N times the bandwidth of a single processor. But effective bandwidth is lower because access to remote memories requires passing messages between one or more intermediate nodes.

   If shared memory is available, the bandwidth depends strongly on the implementation of shared access. A variety of implementations, ranging from a shared bus to a full crossbar, provide a spectrum of performance and cost for the architect to consider. The bus is best suited to systems with few processors, and the shuffle-exchange network, or other similar multilayer interconnection, is an attractive mechanism to use for larger systems because it offers increased performance over the shared bus at a cost that is likely to be commensurate with the performance improvement.

Cache is potentially useful for multiprocessors with a small number of processors. As the number grows to 8, 16, 32, and larger, the cache-coherence problem becomes difficult to solve at reasonable cost. Consequently, caches are likely to be limited in their use to local variables and instructions or in other ways that eliminate the problem of maintaining consistency. Accesses to uncacheable items tend to occupy a disproportionate fraction of memory bandwidth of shared memory and are one of the limiting factors in performance.

Bandwidth is also limited by "hot spots," regions of memory that receive more than their share of accesses. A combining switch reduces the effect of hot spots by reducing the physical data traffic required for concurrent accesses to shared data. Whether or not the combining switch is a cost-effective means for dealing with hot spots is still a matter of intense research, and the outcome of that study may have a profound impact on the future of multiprocessors with hundreds of processors.

3. *Input/output bandwidth:* the multiprocessor provides input/output bandwidth that grows proportionally to the number of processors. To tap the full bandwidth potential, it may be necessary to store data externally in unusual ways. One individual file should be partitioned into multiple segments that can be accessed concurrently by multiple processors, one processor per segment. In general, the multiprocessor offers excellent input/output bandwidth, provided that each processor has independent input/output capability.
4. *Communication bandwidth:* communication bandwidth available within a multiprocessor is strictly a function of the interconnection structure. Bandwidth available through ring and bus interconnections is low in cost, but suitable for systems with up to only 8 or 16 processors. As the number of processors increases above this amount, contention at the communications network tends to degrade performance. To support hundreds or thousands of processors requires a more sophisticated interconnection structure to tie processors to the memory system and to each other.
5. *Synchronization:* multiprocessors without combining networks or the equivalent have a maximum MSYPS rate that is independent of the number of processors, and therefore the maximum sustainable MSYPS rate becomes a serious bottleneck for systems with a moderate to large number of processors.

   The combining switch or the synchronization bus may provide a means for the maximum sustainable MSYPS rate to increase proportionally with the number of processors in a system. The synchronization bus is more attractive because of its lower cost. However, research is still in progress to determine if either or both solutions are cost-effective and practical to implement.

6. *Multiple purpose:* the most versatile parallel processors are multiprocessors because each processor can operate independently of all other processors if this behavior is desirable and all constraints can be satisfied.

This list shows the strengths and weaknesses of multiprocessors. The strengths for multiprocessors are high processing and input/output bandwidths and great flexibility. The weaknesses are synchronization limitations, memory bandwidth, and communication bandwidth. These three areas provide a great challenge for the computer architect because, in an era of fast technological change, new approaches become feasible almost overnight, and old approaches become obsolete as quickly.

Multiprocessors are not as well understood as are vector processors, mainly because their development lagged behind the development of vector processors by more than a decade. In speculating about the future of multiprocessors, we expect to see many systems with a small number of processors. Whether or not the 1000-processor system becomes widely used is only conjectural today and depends strongly on how well new technology can be adapted to the needs of multiprocessors.

Exercises

7.1 The inner loop of an iteration has the following form:

$$A[i] := B[i];$$
$$C[i] := A[i] + B[i-1];$$
$$D[i] := A[i+1];$$

a) Find the precedence constraints among three successive iterations of this loop. which statements depend directly on which statements? Are the individual iterations executable in parallel?

b) Let the middle equation be changed so that $B[i-1]$ becomes $B[i]$. Repeat *a*.

c) Let the middle equation be changed so that $B[i-1]$ becomes $B[i+1]$. Repeat *a*.

7.2 The inner loop of a program is the following:

$$A[i,j] := A[i+1, j-1];$$

a) Let this statement be nested within two loops, the outer loop on i and the inner loop on j. Give an example of loop-control statements that permit the iterations on j to be chunked together and the iterations on i to be independent processes that can be executed in parallel.

b) Give an example of loop-control statements that do not permit independent execution of iterations on j that are chunked together.

7.3 The purpose of this exercise is to explore architectural support for the **do par** phrase. Consider a **do par** loop that is to be repeated N times.

a) Assume a multiprocessor that has access to shared and local memory. Before the **do par** is reached, all program instructions and data are resident in shared memory. Assume that the iterations are truly independent in that there are no READ/WRITE, WRITE/WRITE, or WRITE/READ conflicts. Show a scheme for initializing the iterations so that each iteration can execute concurrently with other iterations, and one copy of the program in shared memory is used for all iterations. Let the index variable for the loop be i and assume that the loop references vector elements $A[i]$ and $B[i]$. To achieve maximum performance, how do you decide whether a datum should be moved to local memory or left in global memory during a loop iteration?

b) The process of initializing and initiating loop iterations can be done sequentially in $O(N)$ time or in parallel in $O(\log N)$ time. Write a brief program suitable for execution in a multiprocessor computer that is capable of initiating 128 iterations of a **do par** loop and has a complexity of $O(\log N)$. Assume the shared and local memory structure used in *a*, and assume that the processes can be initiated immediately and need not be queued while waiting for a processor to become available.

c) Devise some architectural support for the process of *b* to simplify its programming. The support should consist of one or more machine instructions specific to this process. Describe each instruction and the operands that it requires. Describe any other facilities in a multiprocessor architecture required by these instructions to facilitate the initiation process.

7.4 Exercise 7.3 ignores the problem of queueing tasks if processors are unavailable. Assume an $O(\log N)$ task-generation process and consider how to implement task queueing if no processors are available.

a) Assume that the multiprocessor shared memory is accessed via a crossbar switch and that pending tasks are queued on a single-task queue. Develop a performance model that estimates the cost of task queueing and dequeueing under the condition that the number of iterations to run concurrently is twice the number of available processors. How does this change when the number of iterations to run is 1024 times the number of available processors?

b) What specialized instructions for task queueing can assist the process in *a*? Describe what each such instruction does and the operands that it requires. To demonstrate their use, show a program fragment for task queueing that uses these instructions. Include a mechanism for determining whether or not a task has to be queued.

c) Consider an architecture that supports Fetch-and-Add. Repeat *a*.

7.5 The purpose of this exercise is to consider the implementation of the **Barrier** operation. Assume a multiprocessor with shared memory accessed by means of a crossbar switch.

a) Show a sequence of machine instructions that implements the **Barrier** operation. Estimate the machine performance of your code when N processors attempt to execute the code concurrently. Describe why your code works correctly in a concurrent-execution environment.

b) Repeat *a* for a multiprocessor based on a bus interconnection.

c) Repeat *a* for a multiprocessor based on an interconnection network that supports Fetch-and-Add.

7.6 The purpose of this exercise is to compare different synchronization techniques. The objective of the exercise is to create a circular buffer of length *N*. There are two subroutines, *Put* and *Get*, that control input and output to the buffer. The implementation has to be free of deadlock and livelock.

a) Show an implementation of *Put* and *Get* that uses Test-and-Set for synchronization. Use a high-level language plus Test-and-Set to describe your implementation.

b) Repeat *a* using Increment and Decrement instead of Test-and-Set.

c) Repeat *a* using Compare-and-Swap instead of Test-and-Set.

d) Repeat *a* using Fetch-and-Add instead of Test-and-Set.

7.7 The purpose of this exercise is to explore the use of Compare-and-Swap on linked-list implementations of queues.

a) Consider a queue implemented as a linked list with *Head* and *Tail* pointers as described in the body of the chapter. Assume that DEQUEUEs cannot run concurrently with ENQUEUEs and that as many as *N* ENQUEUEs can run concurrently. Give an implementation of ENQUEUE with Compare-and-Swap that works correctly under these conditions, including the ability to add an item to an empty queue.

b) Construct an implementation of DEQUEUE with Compare-and-Swap. How does your implementation handle the special case in which DEQUEUE produces an empty queue? Does your implementation work correctly if run concurrently with ENQUEUE from the first part?

7.8 The purpose of this exercise is to take the reader through the details of a complete and correct implementation of Compare-and-Swap.

a) Examine the skeleton of ENQUEUE as shown in Program 7.9. Note that part of the program is missing. The program does not specify what happens when it tries to place a new value in a link field and discovers that the link field has changed to nonzero. Study this carefully and write a corresponding DEQUEUE program. In your DEQUEUE program you should remove an item from the queue by copying the item in its link field to the *Head* pointer. Note that the instant the *Head* pointer is updated with a 0 value marks an instant in which the ENQUEUE must alter the outcome of its test on the value of *Head*. Your program should install a new link value in a link field of the item deleted that takes on a special value that signifies "Deleted." You should consider writing the DEQUEUE program in either of two ways—one way that modifies the *Head* pointer and then modifies the link field with the "Deleted" value, and the other way that reverses the order.

b) In your DEQUEUE program insert the code that tests for an empty queue, and attempt to set *Tail* to 0 to indicate this condition. Before *Tail* is set to 0, what is the value that should be in this variable if no ENQUEUEs are active? If any ENQUEUES are active, what should be the action of DEQUEUE?

c) Now consider the missing code from Program 7.9 for ENQUEUE. When ENQUEUE discovers a link with a nonzero value that expects to have the value 0, under what condition can this happen? In that case, what code should be executed for ENQUEUE to exit correctly?

d) In either your ENQUEUE or DEQUEUE programs, you may have written a loop that repeatedly tests some variable waiting for another program to alter it. Although this is correct in a technical sense, it is not necessarily a preferred solution. Examine any such loop you have written and determine what function has to be performed after the loop that cannot be performed until the second program takes some action. If the looping program were to exit immediately, the function could be performed instead by the program whose unfinished execution caused the loop to occur. Find a means to eliminate the loop by moving the function to be performed after the loop from one program to another.

e) Reexamine the Compare-and-Swaps in your program. Some or all of them may have to be double Compare-and-Swaps in which one of two items is a counter, as shown in Program 7.9, in order to detect the occurrence of a sequence of events that leaves *Head* or *Tail* or some other variable in a final state that is the same as the initial state. Determine which Compare-and-Swaps must be double (shared variable and a counter variable) and which can be single (no counter variable).

7.9 The purpose of this exercise is to investigate the performance of Dijkstra's shortest path algorithm [1959] on various multiprocessors. The objective is to find the length of the shortest path from Node 1 to Node x for an arbitrarily specified node x in a graph. Dijkstra's algorithm accepts as input N^2 point-to-point distances among N nodes. Let the distances be given in the matrix $D[i, j]$. The matrix is symmetric and all entries are nonnegative. The algorithm is a node-labeling algorithm in which nodes are initially given temporary labels that give an upper bound on the shortest path to each node. At the end of each major iteration, some temporary label becomes permanent and never changes again in the course of the algorithm. Eventually all labels are made permanent, at which point the algorithm has found the length of the shortest path from Node 1 to any other node in the graph. Labels are held in the array L.

Write a parallel code for the following algorithm and find its complexity.

a) Give Node 1 the permanent label 0, that is, set $L[1]$ to 0.

b) Label Nodes 2 through N with temporary labels such that $L[i]$, the label for Node i, receives the value $D[1, i]$. (The distance to Node i is not greater than $D[1, i]$.)

c) Among the temporary labels, find the node with the smallest label, breaking ties arbitrarily. Let this be Node j. Make this label permanent.

d) For each node with a temporary label, such as Node k, change its label to $L[j] + D[j, k]$ if that is less than its current label. (The shortest path to Node j, followed by the direct path from Node j to Node k, is shorter than the best path to Node k found thus far.)

7.10 *QuickSort* is a very fast sorting algorithm that can be described succinctly by the following Pascal-like program.

```
Procedure QuickSort (Low: integer, High: integer, var A array of real);
{* Sort the array A for the range starting at Low and ending at High *}
begin
  var Pivot: integer;
    Procedure Partition(Low,High,A,Pivot);
  {* Partition guesses the median of the numbers in the array between Low and
    High, then moves data around in the array so that A[Pivot] contains its guess
    for the median, and the indices between Low and Pivot − 1 contain smaller
    values than A[Pivot], and the indices between Pivot + 1 and High contain
    larger values than A[Pivot]. *}
    begin
      ...
    end; {* of Partition *}
  if Pivot − Low > 1 then QuickSort(Low,Pivot − 1,A);
  if High − Pivot > 1 then QuickSort(Pivot + 1,High,A);
end;
```

We wish to run this program on a multiprocessor.

a) Modify the algorithm in some fashion to exploit multiprocessing. Use a high-level language to describe your multiprocessing version of the algorithm, and explain in English how your algorithm functions.

b) Describe the architecture of a multiprocessor that executes your algorithm. If your architecture passes messages among processors when a parallel procedure is invoked, indicate how much information is passed for a call on *QuickSort*. If your processor has a shared memory, describe how many references occur to shared memory at the point of calling a parallel procedure, and count these references for the case of a call to *QuickSort*.

c) Hand simulate the execution of *QuickSort* on your architecture for a small example. In this example, what are the bottlenecks for a multiprocessor implementation?

d) Assume that *Partition* fortuitously always finds the median in its assigned region of an array, and that it does so in time proportional to the size of that portion of the array. Then what is the asymptotic complexity of the *QuickSort* problem on your architecture? Show your derivation.

7.11 The intent of this question is to explore the effectiveness of parallel search in an AND/OR search tree.

a) Consider a very simple OR search tree that consists of M alternatives, any one of which might lead to a satisfactory solution for a search. Assume that each alternative has a probability p of being satisfactory, and q for being unsatisfactory, and the alternatives are independent. A serial search of this tree completes the exploration of one alternative during a single step, and halts when the first success is discovered or when all alternatives are exhausted.

If N processors are used to search the tree in parallel, they take a single step to search N of the possible alternatives, assuming that $N \leq M$. What is the speedup for $p = 0.1$? For $p = 0.9$? Why is the speedup dependent on p? Give an intuitive explanation for your answer.

b) Repeat the first part for a similar tree whose root node is an AND node instead of an OR node. That is, the tree search is successful only if all alternatives succeed, otherwise it is unsuccessful. A serial search terminates when the first unsuccessful alternative is found, or if the entire tree is searched and all alternatives are successful.

c) Consider a two-level tree whose root node is a two-alternative AND node and whose nodes at the next level are two-alternative OR nodes. Let p be the probability of success of an OR-node alternative. For small values of p, what is the potential speedup of a parallel search and how do you schedule processors to achieve this speedup? For large values of p, what is the potential speedup, and how do you schedule processors to achieve this speedup?

d) Finally, consider multilevel trees, with all nodes having two successors, and with nodes at successive levels alternating between OR nodes and AND nodes. (The top node is an AND node; its offspring are OR nodes; their offspring are AND nodes, ...). If the tree has M levels, each node with two offspring, then the number of leaf nodes is 2^M. The potential parallelism thus is 2^M. If p is the probability of success of a leaf node, show that the best possible parallelism is $O(\sqrt{M})$ for large and small p. What happens for p that are near the center of the range?

References

Agarwal, A., J. Hennessy, and M. Horowitz. "An analytical cache model." *ACM Transactions on Computer Systems*, **7**, no. 2, 184–215, May 1989.

Agerwala, T., and J. Cocke. "High performance reduced instruction set processors." IBM Research Division Report RC 12434, March 31, 1987.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.

Allen, J. R. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*, Ph.D. thesis, Rice University, 1983.

Allen, J. R., K. Kennedy, C. Porterfield, and J. Warren. "Conversion of control dependence to data dependence." *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Tex., January 1983.

Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr. "Architecture of the IBM System/360." *IBM Journal of Research and Development*, **8**, no. 2, 87–101, April 1964.

Archibald, J., and J.-L. Baer. "Cache coherence protocols: Evaluation using a multiprocessor simulation model." *ACM Transaction on Computers*, **4**, no. 4, 273–298, November 1986.

Baer, J.-L. *Computer Systems Architecture*. Potomac, Md.: Computer Science Press, 1980.

Batcher, K. E. "Sorting networks and their applications." *AFIPS Conference Proceedings, 1968 SJCC*, **32**, Washington, D.C.: Thompson Books, 307–314, 1968.

Beetem, J., M. Denneau, and D. Weingarten. "The GF-11 supercomputer." *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Cat. No. 85CH2140-2, 108–115, August 1985.

Belady, L. "A study of replacement algorithms for a virtual-store computer." *IBM Systems Journal*, **5**, no. 2, 78–101, 1966.

Bell, C. G., and A. Newell. *Computer Structures: Readings and Examples*. New York: McGraw-Hill, 1971.

Benes, V. "Optimal rearrangeable multistage connecting networks." *Bell System Technical Journal*, **43**, no. 4, 1641–1656, July 1964.

Booth, A. D. "A signed binary multiplication technique." *Quarterly Journal of Mech. Appl. Math.*, **4**, part 2, 1951.

Boral, H., and D. J. DeWitt. "Database machines: An idea whose time has passed? A critique of the future of database machines." In *Database Machines*, edited by H. O. Leilich and M. Missikoff, Berlin: Springer Verlag, 166–187, 1983.

Brunk, H. D. *An Introduction to Mathematical Statistics*. Boston: Ginn and Co., 1960.

Budnik, P. P., and D. J. Kuck. "The organization and use of parallel memories." *IEEE Transactions on Computers*, **C-20**, no. 12, 1566–1569, 1971.

Burks, A. W., H. H. Goldstine, and J. von Neumann. "Preliminary discussion of the logical design of an electronic computing instrument." *U. S. Army Ordnance Department Report*, 1946. Reprinted in Bell and Newell [1971], 92–119.

Buzbee, B. L., G. H. Golub, and C. W. Nielson. "On direct methods for solving Poisson's equation." *SIAM Journal of Numerical Analysis*, **7**, 627–656, 1970.

Charlesworth, A. E., and J. L. Gustafson. "Introducing replicated VLSI to supercomputing: the FPS-164/MAX scientific computer." *Computer*, **19**, no. 3, 10–23, March 1986.

Chen, P. Y., D. H. Lawrie, P. C. Yew, and D. A. Padua. "Interconnection networks using shuffles." *Computer*, **14**, no. 12, 55–64, December 1981.

Chen, T. C. "Parallelism, pipelining, and computer efficiency." *Computer Design*, 69–74, January 1971.

Chen, T. C. "Overlap and pipeline processing." Chapter 9 of *Introduction to Computer Architecture*, edited by H. Stone, Chicago: Science Research Assoc., 427–486, 1980.

Chu, W. W., and H. Opderbeck. "Program behavior and the page-fault-frequency replacement algorithm." *Computer*, **9**, no. 11, 29–38, November 1976.

Clark, D. W, and J. S. Emer. "Performance of the VAX-11/780 translation buffer: simulation and measurement." *ACM Transactions on Computer Systems*, **3**, no. 1, 31–62, February 1985.

Cocke, J., and V. Markstein. "The evolution of RISC technology at IBM." *IBM Journal of Research and Development*, **34**, no. 1, 4–11, January 1990.

Coffman, E. G., Jr., and P. J. Denning. *Operating Systems Theory*. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

Colwell, R. P., *et al.* "Computers, complexity, and controversy." *Computer*, **18**, no. 9, 8–19, September 1985.

Cooley, J. W., and J. W. Tukey. "An algorithm for the machine calculation of complex Fourier series." *Mathematics of Computation*, **19**, 297–301, April 1965.

Coonen, J. T. "An implementation guide to a proposed standard for floating-point arithmetic." *Computer,* **13**, no. 1, 68–79, January 1980.

Crowther, W., *et al.* "Performance measurements on a 128-node butterfly parallel processor." *Proceedings of the 1985 International Conference on Parallel Processing,* IEEE Cat. No. 85CH2140-2, 531–540, August 1985.

Cvetanovic, Z. *Performance Analysis of Multiple-Processor Systems,* Ph.D. thesis, ECE Department, University of Massachusetts, 1985.

Cvetanovic, Z. "Performance analysis of the FFT algorithm on a shared-memory parallel architecture." *IBM Journal of Research and Development,* **31**, no. 4, 435–451, July 1987.

Cytron, R. G. *Compile-Time Scheduling and Optimization for Asynchronous Machines,* Ph.D. thesis, Univ. of Illinois, 1984.

Davidson, E. S. "The design and control of pipelined function generators." *Proceedings of the 1971 International Conference on Systems, Networks, and Computers,* Oaxtepec, Mexico, 19–21, January 1971.

Denning, P. J. [1968a]. "Thrashing: Its causes and prevention." *AFIPS Conference Proceedings, 1968 FJCC,* **33**, Washington, D.C.: Thompson Books, 915–922, 1968.

Denning, P. J. [1968b]. "The working-set model for program behavior." *Communications of the ACM,* **11**, no. 5, 323–333, May 1968.

Denning, P. J., J. E. Savage, and J. R. Spirn. "Models for locality in program behavior." Department of Electrical Engineering, Princeton Univ., Princeton, New Jersey Computer Science Report TR-107, April 1972.

Dias, D. M., and J. R. Jump. "Packet switching interconnection networks for modular systems." *Computer,* **14**, no. 12, 43–54, December 1981.

Dijkstra, E. W. "A note on two problems in connexion with graphs." *Numerishce Mathematik,* **1**, 269–271, 1959.

Dijkstra, E. W. "Solution of a problem in concurrent programming." *Communications of the ACM,* **8**, 569–570, September 1965.

Ditzel, D. R., and H. R. McLellan. "Branch folding in the CRISP microprocessor: Reducing branch delay to zero." *Proceedings of the 14th International Symposium on Computer Architecture,* IEEE Cat No. 87CH2420-8, Pittsburgh, Pa., 2–9, June 1987.

Dubois, M., and F. A. Briggs. "Effects of cache coherency in multiprocessor systems." *IEEE Transactions on Computers,* **C-31**, no. 11, 1083–99, November 1982.

Dubois, M., C. Scheurich, and F. Briggs. "Memory access buffering in multiprocessors." *Proceedings of the 13th International Symposium on Computer Architecture,* Tokyo, Japan, 434–442, June 1986.

Eckhouse, R. H., Jr., and H. M. Levy. *Computer Programming and Architecture: The VAX-11.* Bedford, Mass.: Digital Press, 1980.

Eggers, S. J., and R. H. Katz. "Evaluating the performance of four snooping cache coherency protocols." *Proceedings of the 16th International Symposium on Computer Architecture,* IEEE Catalog Number 89CH2705-2, 2–15, June 1989.

Flynn, M. J. "Very high-speed computers." *Proceedings of the IEEE,* **54**, 1901–1909, December 1966.

Ford, L. R., Jr., and D. R. Fulkerson. "Maximal flow through a network." *Canadian Journal of Mathematics*, **8**, 399–404, 1956.

Forsythe, G., and C. B. Moler. *Computer Solution of Linear Algebraic Systems*. Englewood Cliffs, N.J.: Prentice-Hall, 1967.

Fox, G., *et al*. *Solving Problems on Concurrent Processors, Vol. 1, General Techniques and Regular Problems*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

Ghanem, M. Z. "Dynamic partitioning of the main memory using the working set concept." *IBM Journal of Research and Development*, **19**, no. 5, 445–450, September 1975.

Gharachorloo, K., *et al*. "Memory consistency and event ordering in scalable shared-memory multiprocessors." *Proceedings of the 17th International Symposium on Computer Architecture*, IEEE Cat No. 90CH2887-8, Seattle, Wash., 16–26, May 1990.

Goodman, J. "Using cache memory to reduce processor-memory traffic." *Proceedings of the 10th International Symposium on Computer Architecture*, Stockholm, Sweden, 124–131, June 1983.

Gottlieb, A., *et al*. "The NYU Ultracomputer–Designing an MIMD shared-memory parallel computer." *IEEE Transactions on Computers*, **C-32**, no. 2, 175–189, February 1983.

Grohoski, G. F. "Machine organization of the IBM RISC System/6000 processor." *IBM Journal of Research and Development*, **34** no. 1, 37–58, January 1990.

Green, P., Jr., and H. S. Stone. "The implementation of a barrier for multiprocessors by means of an optical bus." *IBM Technical Disclosure Bulletin*, **33**, no. 1B, 291–292, 2 June 1990.

Halstead, R. "Multilisp: An overview and working example." *ACM Transactions on Programming Languages and Systems*, **7**, no. 4, 501–538, October 1985.

Hayes, J. P. *Computer Architecture and Organization*. New York: McGraw-Hill, 1978.

Heidelberger, P., B. D. Rathi, and H. S. Stone. "A low-cost contention-free barrier synchronization." *IBM Technical Disclosure Bulletin*, **31**, no. 11, 328–329, 2 April 1989.

Heidelberger, P., B. D. Rathi, and H. S. Stone. "A device for performing efficient task-distribution with a bus connection." *IBM Technical Disclosure Bulletin*, **32**, no. 9A, 360–362, 2 January 1990.

Heller, D. E. "Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems." *SIAM Journal of Numerical Analysis*, **13**, 484–496, 1976.

Heller, D. E. "A survey of parallel algorithms in numerical linear algebra." *SIAM Review*, **20**, no. 4, 740–777, 1978.

Hennessy, J. L., and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, San Mateo, Calif.: Morgan Kaufmann, 1990.

Hill, M., *et al*. "Design decisions in SPUR." *Computer*, **19**, no. 11, 8–22, November 1986.

Hill, M., and A. Smith. "Evaluating associativity in CPU Caches." *IEEE Transactions on Computers*, **38**, no. 12, 1612–1630, December 1989.

Hillis, W. D. *The Connection Machine*. Cambridge, Mass.: MIT Press, 1986.

Hillis, W. D., and G. L. Steele, Jr. "Data parallel algorithms." *Communications of the ACM*, **29**, no. 12, 1170–1184, December 1986.

Hoshino, T. "Invitation to the world of 'PAX'." *Computer*, **19**, no. 5, 68–79, May 1986.

Hoshino, T. *PAX Computer: High-Speed Parallel Processing and Scientific Computing*. Reading, Mass.: Addison-Wesley, 1989.

Hwang, K. *Computer Arithmetic: Principles, Architecture and Design*. New York: Wiley, 1978.

Hwang, K., and F. A. Briggs. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

IBM System/370 Principles of Operation, GA22-7000-9, File No. S370-01, Tenth Edition, 1983.

IEEE. *IEEE Standard 754–1985 for Binary Floating-Point Arithmetic*. Order No. CN953, 1985.

Indurkhya, B., H. S. Stone, and L. Xi-Cheng. "Optimal partitioning of randomly generated distributed programs." *IEEE Transactions on Software Engineering*, **SE-12**, no. 3, 483–495, March 1986.

James, D. V., *et al.* "Scalable coherent interface." *Computer*, **23**, no. 6, 74–77, June 1990.

Jouppi, N. P. "The nonuniform distribution of instruction-level and machine parallelism and its effect on performance." *IEEE Transactions on Computers*, **38**, no. 12, 1645–1658, December 1989.

Karp, R. M., and W. L. Miranker. "Parallel minimax search for a maximum." *Journal of Combinatorial Theory*, **4**, no. 1, 19–39, 1968.

Kilburn, T. D., R. B. Payne, and D. J. Howarth. "One-level storage system." *IRE Transactions on Electronic Computers*, **EC-11**, no. 2, 223–235, April 1962.

Kobayashi, M., and M. H. MacDougall. "The stack growth function: cache line reference models." *IEEE Transactions on Computers*, **C-38**, no. 6, 798–804, June, 1989.

Kogge, P. M. *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.

Kogge, P. M., and H. S. Stone. "A parallel algorithm for the efficient solution of a general class of recurrence equations." *IEEE Transactions on Computers*, **C-22**, 786–93, 1973.

Kruskal, C. P., and M. Snir. "The performance of multistage interconnection networks for multiprocessors." *IEEE Transactions on Computers*, **C-32**, 1091–1098, December 1983.

Kuck, D. J. "Parallel processing in ordinary programs." In *Advances in Computers*, **15**, edited by Rubinoff and Yovits, New York: Academic Press, 119–179, 1976.

Kuck, D. J., R. H. Kuhn, B. Leasure, and M. Wolf. "The structure of an advanced vectorizer for pipelined programs." In *Tutorial on Supercomputers: Designs and Applications*, edited by K. Hwang, New York: IEEE Press EH0219-6, 163–178, 1984.

Kuck, D. J., Y. Muraoka, and S.-C. Chen. "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup." *IEEE Transactions on Computers*, **C-21**, no. 12, 1293–1310, December 1972.

Kung, H. T., and C. E. Leiserson. "Systolic arrays (for VLSI)." In *1978 Symposium on Sparse Matrix Computations and Their Applications*, edited by I. S. Duff and G. W. Stewart, 48–53, 1978.

Laha, S., J. H. Patel, and R. K. Iyer. "Accurate low-cost methods for performance evaluation of cache memory systems." *IEEE Transactions on Computers*, **C-37**, no. 11, 1325–1336, November 1988.

Lamport, L. "The parallel execution of DO loops." *Communications of the ACM*, **17**, no. 2, 83–93, February 1974.

Lamport, L. "How to make a multiprocessor computer that correctly executes multiprocess programs." *IEEE Transactions on Computers*, **C-28**, no. 9, 690–691, September 1979.

Lawler, E. L. *Combinatorial Optimization: Networks and Matroids*, New York: Holt, Rinehart, and Winston, 1976.

Lawrie, D. H. "Access and alignment of data in an array processor." *IEEE Transactions on Computers*, **C-24**, 496–503, December 1975.

Lenoski, D., *et al.* "The directory-based cache coherence protocol for the DASH multiprocessor." *Proceedings of the 17th International Symposium on Computer Architecture*, IEEE Cat No. 90CH2887-8, Seattle, Wash., 149–159, May 1990.

Losq, J. J., G. S. Rao, and H. E. Sachar. "Decode history table for conditional branch instructions." U. S. Patent No. 4,477,872, October 1984.

Mashburn, H. H. "The C.mmp/Hydra project: an architectural overview." Chapter 22 of *Computer Structures: Principles and Examples*. D. P. Siewiorek, C. G. Bell, and A. Newell, New York: McGraw-Hill, 350–370, 1982.

Mattson, R. L., J. Gecsei, D. R. Slutz, and I. L. Traiger. "Evaluation techniques for storage hierarchies." *IBM Systems Journal*, **9**, 78–117, 1970.

Mead C., and L. Conway. *Introduction to VLSI Systems*. Reading, Mass.: Addison-Wesley, 1980.

Miura, K. "Vectorization of phase space Monte Carlo code in FACOM vector processor VP-200." *Proceedings of the 1985 Conference on Computing in High Energy Physics*, edited by L. O. Hertzberger and W. Hoogland, Amsterdam: North-Holland, Elsevier, 401–408, 1986.

Nicol, D. M. "Optimum partitioning of random programs across two processors." *IEEE Transactions on Software Engineering*, **SE-15**, no. 2, 134–141, February 1989.

Organick, E. I. *The Multics System: An Examination of Its Structure*. Cambridge, Mass.: MIT Press, 1972.

Organick, E. I. *Computer System Organization: the B5700/B6700 Series*, New York: Academic Press, 1973.

Padmanabhan, K., and D. H. Lawrie. "Performance analysis of redundant-path networks for multiprocessor systems." *ACM Transactions on Computer Systems*, **3**, no. 2, 117–144, May 1985.

Padua, D. A., and M. J. Wolfe. "Advanced compiler optimizations for supercomputers." *Communications of the ACM*, **29**, no. 12, 1184–1201, December 1986.

Patel, J. H., and E. S. Davidson. "Improving the throughput of a pipeline by insertion of delays." *Proceedings of the Third Annual Computer Architecture Symposium*, IEEE No. 76CH 0143-5C, 159–163, 1976.

Patt, Y., W.-M. Hwu, and M. Shebanow. "HPS, a new microarchitecture: Rationale and introduction." *Proceedings of the 18th Annual Workshop on Microprogramming*, IEEE Computer Society Press, 103–108, December 1985.

Patterson, D. A. "Reduced instruction set computers." *Communications of the ACM*, **28,** no. 1, 8–21, January 1985.

Patterson, D. A., and C. H. Sequin. "A VLSI RISC." *Computer*, **15,** no. 9, 8–21, September 1982.

Pease, M. C. "An adaptation of the fast Fourier transform for parallel processing." *Journal of the ACM*, **15,** 252–264, 1968.

Pfister, G., *et al.* "The IBM Research Parallel Prototype (RP3): Introduction and architecture." *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Cat. No. 85CH2140-2, 764–771, August 1985.

Pfister, G., and V. A. Norton. " 'Hot Spot' contention and combining in multistage interconnection networks." *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Cat. No. 85CH2140-2, 790–795, August 1985.

Pomerene, J., T. R. Puzak, R. Rechtschaffen, and F. Sparacio. "Prefetching mechanism for a high-speed buffer store." Patent Pending, 1984.

Preparata, F., and J. Vuillemin. "The cube-connected cycles: A versatile network for parallel computation." *Communications of the ACM*, **25,** 300–309, 1981.

Puzak, T. R. *Cache-Memory Design*, Ph.D. thesis, ECE Department, University of Massachusetts, 1985.

Radin, G. "The 801 minicomputer." *Proceedings of the Symposium for Programming Languages and Operating Systems Support*, 39–47, 1982.

Salton, G., and C. Buckley. "Parallel text search methods." *Communications of the ACM*, **31,** no. 2, 202–215, February 1988.

Scheurich, C., and M. DuBois. "Correct memory operation of cache-based multiprocessors." *Proceedings of the 14th International Symposium on Computer Architecture*, IEEE Cat No. 87CH2420-8, Pittsburgh, Pa., 234–243, June 1987.

Seitz, C. L. "The Cosmic Cube." *Communications of the ACM*, **28,** no. 1, 22–33, January 1985.

Shar, L. E., and E. S. Davidson. "A multiminiprocessor system implemented through pipelining." *Computer*, **7,** no. 2, 42–51, February 1974.

Shemer, J. E., and S. C. Gupta. "On the design of Bayesian storage allocation algorithms for paging and segmentation." *IEEE Transactions on Computers*, **C-18,** no. 7, 644–651, July 1969.

Shemer, J. E., and B. Shippey. "Statistical analysis of paged and segmented computer systems." *IEEE Transactions on Electronic Computers*, **EC-15,** no. 6, 855–863, December 1966.

Siewiorek, D. P., C. G. Bell, and A. Newell. *Computer Structures: Principles and Examples*. New York: McGraw-Hill, 1982.

Singh, J. P., H. S. Stone, and D. F. Thiebaut. "An analytical model for fully associative cache memories." *IEEE Transactions on Computers*, **41**, no. 7, 811–825, July 1992.

Singleton, R. C. "On computing the fast Fourier transform." *Communications of the ACM*, **10**, 647–654, 1967.

Sites, R. "Operating systems and computer architecture." Chapter 12 of *Introduction to Computer Architecture*, 2nd ed., edited by H. S. Stone, Chicago: Science Research Associates, 1980.

Slotnick, D. L., W. C. Borck, and R. C. McReynolds. "The SOLOMON computer." *AFIPS 1962 Fall Joint Computer Conference*, **22**, Washington, D.C.: Spartan Books, 97–107, 1962.

Smith, A. "Cache memories." *ACM Computing Surveys*, **14**, no. 3, 473–530, September 1982.

Smith, A. "Cache evaluation and the impact of workload choice." *Proceedings of the 12th Annual Computer Architecture Symposium*, IEEE No. 85CH2 144–4, 64–75, June 1985.

Smith, A. "Line (block) size choice for CPU cache memories." *IEEE Transactions on Computers*, **C-36**, no. 9, 1063–1075, September 1987.

Smith, D. R. "Random trees and the analysis of branch and bound procedures." *Journal of the ACM*, **31**, no. 1, 163–188, January 1984.

Smith, J. E., and J. R. Goodman. "Instruction cache replacement policies and organizations." *IEEE Transactions on Computers*, **C-34**, no. 3, 234–241, March 1985.

Sohi, G. S., J. E. Smith, and J. R. Goodman. "Restricted Fetch & Φ Operations for parallel processing." *Proceedings of the 3rd International Conference on Supercomputing*, Crete, Greece, 410–416, June, 1989.

Sterbenz, P. H. *Floating-Point Computation*. Englewood Cliffs, N.J.: Prentice-Hall, 1974.

Stanfill, C., and B. Kahle. "Parallel free-text search on the Connection Machine system." *Communications of the ACM*, **29**, no. 12, 1229–1239, December 1986.

Stone, H. S. "Parallel processing with the perfect shuffle." *IEEE Transactions on Computers*, **C-20**, 153–161, 1971.

Stone, H. S. "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations." *Journal of the ACM*, **20**, 27–38, January 1973.

Stone, H. S. "Database applications of the Fetch-and-Add instruction." *IEEE Transactions on Computers*, **C-33**, no. 7, 604–612, July 1984.

Stone, H. S. "Parallel querying of large databases: A case study." *Computer*, **20**, no. 10, 11–21, October 1987.

Stone, H. S., ed. *Introduction to Computer Architecture*. Chicago, Ill.: Science Research Associates, 1974.

Stone, H. S., ed. *Introduction to Computer Architecture*, 2nd ed. Chicago, Ill.: Science Research Associates, 1980.

Stone, H. S., J. L. Wolf, and J. Turek. "Optimal partitioning of cache memory." *IEEE Transactions on Computers*, **41**, no. 1054–1068, 1992.

Strecker, W. D. "Transient behavior of cache memories." *ACM Transactions on Computer Systems*, **1**, no. 4, 281–293, November 1983.

Sullivan, H., T. Bashkow, and D. Klappholtz. "A large-scale homogeneous fully distributed parallel machine." *Proceedings of the Fourth Annual Symposium on Computer Architecture*, 105–124, 1977.

Sussenguth, E. "Instruction sequence control." U. S. Patent No. 3,559,183, January 26, 1971.

Sweazey, P., and A. J. Smith. "A class of compatible cache-consistency protocols and their support by the IEEE Futurebus." *Proceedings of the 13th International Symposium on Computer Architecture*, Tokyo, Japan, 414–423, June 1986.

Tanenbaum, A. S. *Structured Computer Organization*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.

Thanawastien, S., and V. P. Nelson. "Interference analysis of shuffle/exchange networks." *IEEE Transactions on Computers*, **C-30**, 545–556, August 1981.

Thiebaut, D. F. "On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio." *IEEE Transactions on Computers*, **C-38**, no. 7, 1012–1026, July 1989.

Thiebaut, D. F., and H. S. Stone. "Footprints in the cache." *ACM Transactions on Computer Systems*, **5**, no. 4, 305–329, November 1987.

Thiebaut, D. F., H. S. Stone, and J. L. Wolf. "Improving disk cache hit-ratios through cache partitioning." *IEEE Transactions on Computers*, **41**, no. 6, 665–676, June 1992.

Thompson, D. "Multi-device apparatus synchronized to the slowest device." U. S. Patent No. 4, 493,053, January 8, 1985.

Thompson, J. G., and A. J. Smith. "Efficient (stack) algorithms for analysis of write-back and sector memories." *ACM Transactions on Computer Systems*, **7**, no. 1, 78–116, February 1989.

Thornton, J. E. *Design of a Computer: The Control Data 6600*, Glenview, Ill: Scott, Foresman, 1970.

Tomasulo, R. M. "An efficient algorithm for exploiting multiple arithmetic units." *IBM Journal of Research and Development*, **11**, no. 1, 25–33, January 1967.

Treiber, R. K. "Systems programming: Coping with parallelism." IBM Research Report RJ 5118, IBM T. J. Watson Research Center, April 1986.

Trivedi, K. S. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Englewood Cliffs, N.J.: Prentice-Hall, 1982.

Varga, R. S. *Matrix Iterative Analysis*. Englewood Cliffs, N.J.: Prentice-Hall, 1962.

Voldman, J., and L. W. Hoevel. "The software-cache connection." *IBM Journal of Research and Development*, **25**, no. 6, 877–893, November 1981.

Voldman, J., *et al*. "Fractal nature of software-cache interaction." *IBM Journal of Research and Development*, **27**, no. 2, 164–170, March 1983.

Wallace, C. C. "A suggestion for a fast multiplier." *IEEE Transactions on Electronic Computers*, **EC-13**, 14–17, 1964.

Wang, W.-H., and J.-L. Baer. "Efficient trace-driven simulation methods for cache performance analysis." *ACM Transactions on Computer Systems*, **9**, no. 3, 222–241, August 1991.

Waser, S., and M. J. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. New York: CBS College Publishing, 1982.

Wilkes, M. V. "Slave memories and dynamic storage allocation." *IEEE Transactions on Electronic Computers*, **EC-14**, no. 2, 270–271, 1965.

Yew, P.-C., D. A. Padua, and D. H. Lawrie. "Stochastic properties of a multiple-layer single-stage shuffle-exchange network in a message-switching environment." *Journal of Digital Systems*, **VI**, no. 4, 387–410, 1983.

Index and Glossary