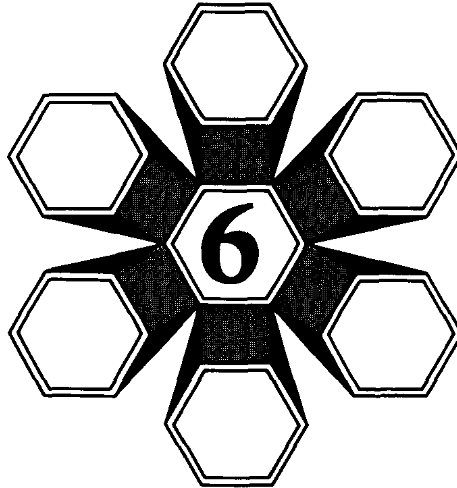
Homayoun

Reference 42



Multiprocessors

6.1 Background
6.2 Multiprocessor Performance
6.3 Multiprocessor Interconnections
6.4 Cache Coherence in Multiprocessors
6.5 Summary

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 towards 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.

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 2

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 pro-

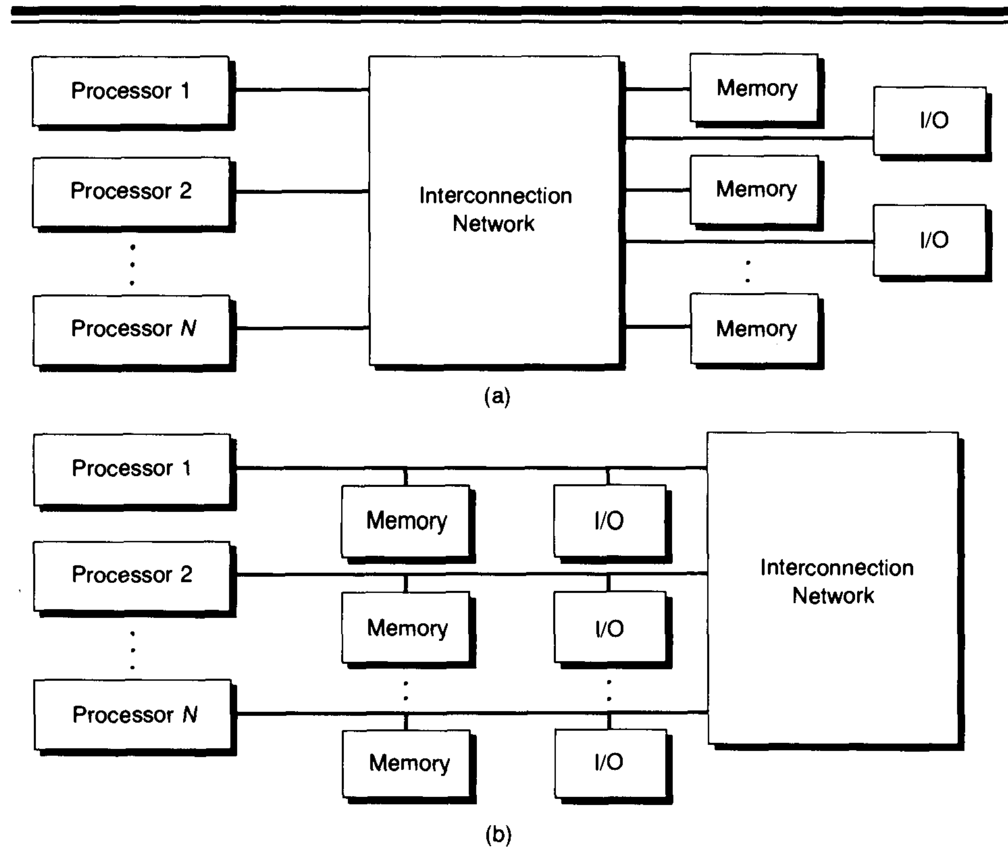


Fig. 6.1 Two multiprocessor structures:
(a) All memory and I/O are remote and shared; and
(b) All memory and I/O are local and private.

cessors. No sharing of memory and input/output is permitted in Fig. 6.1(b). In both 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. Fig. 6.1 shows the extremes in the

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 4

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 ten percent of the peak rate, then ten processors are required in a multiprocessor system just to do the work of a single processor.

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 5

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 from one to six processor systems 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 ten, 1,000, or one million 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 benefits 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.

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 7

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 course 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 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 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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 8

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:

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

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 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(a) shows that this term looks like the letter V because it is symmetric at 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 $M/2$.

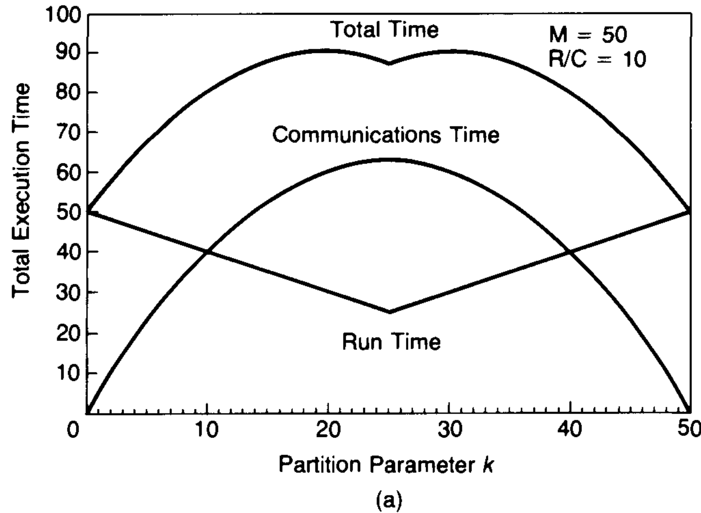
In Fig. 6.2(b), the minimum occurs at $k=0$. The minimum has to be at an extreme point in the region $0 \le k \le M/2$ because the quadratic curve $k(M-k)$ is concave downward, and, after adding a linear term to this curve, the concavity is unchanged. A curve that is concave downward has its minimum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

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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 10



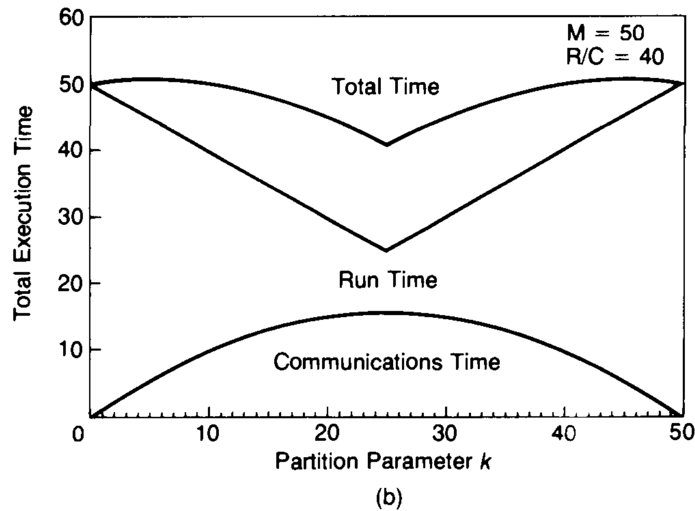


Fig. 6.2 Parallel execution time for two different R/C ratios:
(a) Optimum partition parameter $k = 0$; and
(b) Optimum partition parameter $k = M/2$.

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 four tasks on four processors and three 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

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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 12

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

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

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

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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 14

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 two or more summations have almost the same number of terms, and this number is maximum among all equations, 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 another summation, but the values of summations will be fairly close, so that Eq. (6.7) is approximately if not exactly correct.

Nicol [1986] 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 with 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

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

$$= \frac{RM}{N(N+1)} - C$$

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 ten processors.

This model differs from our original model in the second term. In the original model the cost of the second term grows quadratically with the constant M and diminishes inversely with N. The dependence on N is due to

the reduction in overhead when N things are grouped together on one processor. Because both the first and second terms grow smaller with N, 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

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

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

$$R(M - k) = C(M - k)k \tag{6.12}$$

which occurs at

$$k = \frac{R}{C} \tag{6.13}$$

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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 17

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 towards 1, the speedup falls off towards 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 (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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 18

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

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

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 is in part due to the fact that early finishers have to wait for late finishers.

The first term of Eq. (6.17) tends to decrease inversely with N, and the second term tends to increase linearly with N, which is a situation studied earlier in this section. The first term is minimized by an even distribution of tasks to processors, but this is offset by an increase in the second term.

We know that for any N, Eq. (6.17) 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

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

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

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

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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 21

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 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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 22

multiprocessors. Our analyses in this section strongly suggest that efficiency becomes limited as the number of processors increases. Perhaps an architecture with four 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.

Efficiency is clearly a major concern in the design of multiprocessors. A design that uses $2N$ 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;
- Multilevel switched-network systems;
- Hypercubes; and
- Crossbar-connected systems.

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.

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,

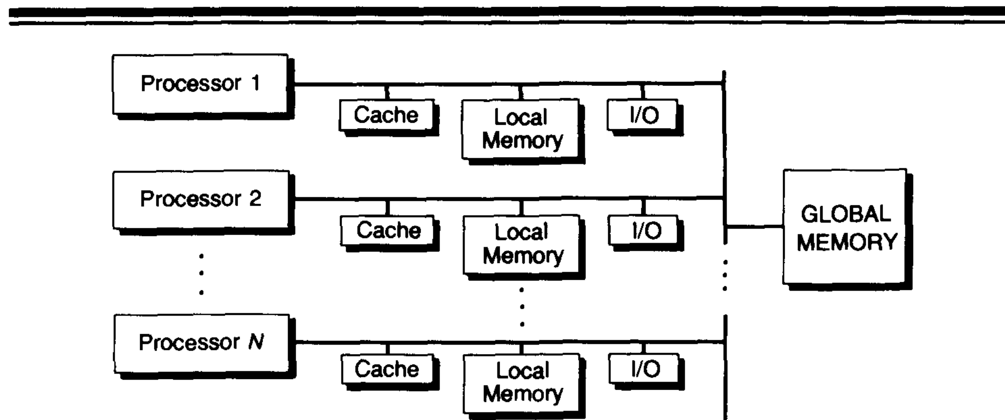


Fig. 6.3 A bus-connected multiprocessor.

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 ten 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.

We expect that bus-oriented systems can support ten processors efficiently and possibly can be stretched to 20 or 30. Beyond this range, bus contention leads to degraded performance to the extent that such systems are unlikely to support 1000 processors or more unless a technological breakthrough provides very high bus bandwidth at very low cost. Even then, such a breakthrough may simply shift the bottleneck from the bus to the shared

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 24

memory. Shared memory, too, is a source of contention different from the bus, and shared-memory may well saturate at, for example, 100 processors, even when the bus bandwidth can support 1000 processors.

There are special issues involved in using caches in a bus-oriented 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, local memory or shared memory.

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.

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.

If exotic technology is used 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 gains due to its use are 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. A possibility for the future is to use optical links 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 ten super-technology processors can do the work of 1000 low-technology processors with a super-technology bus.

The ten-processor, all-super-technology system might well be more cost-effective than the 1000-processor system because it is more 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 in-

creases, performance is eventually limited by the bandwidth of the shared memory. Because processes synchronize their activities by reading and writing shared memory cells, as the number of processes 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. Processes wait at the barrier until all have reached it. Then they are free to continue. The barrier cell can be initialized to the value N, and, as each process reaches it, the cell is decremented. When the cell is decremented to 0, all processes 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 processes 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. Unfortunately, there is no guarantee that any of the approaches will succeed.

Returning to the bus interconnection, 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.

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 26

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.

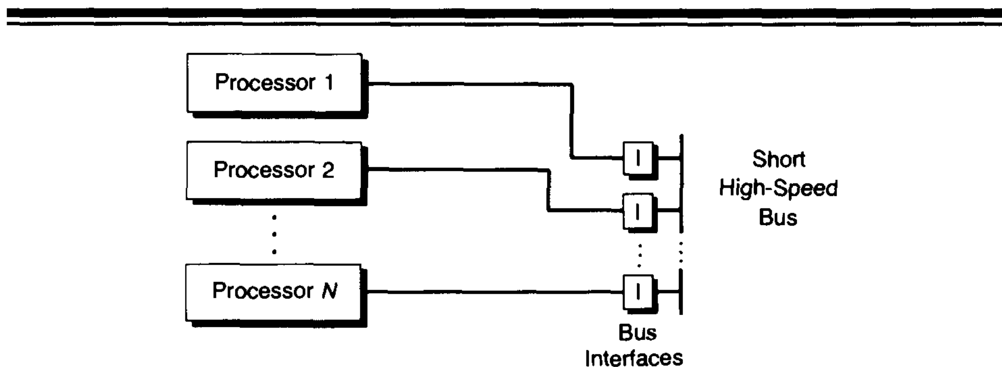


Fig. 6.4 A high-speed bus with a short physical length connecting a collection of processors. The I-unit is an interface that permits processors to be relatively far from the bus when compared to the physical length of the bus itself.

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.

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.

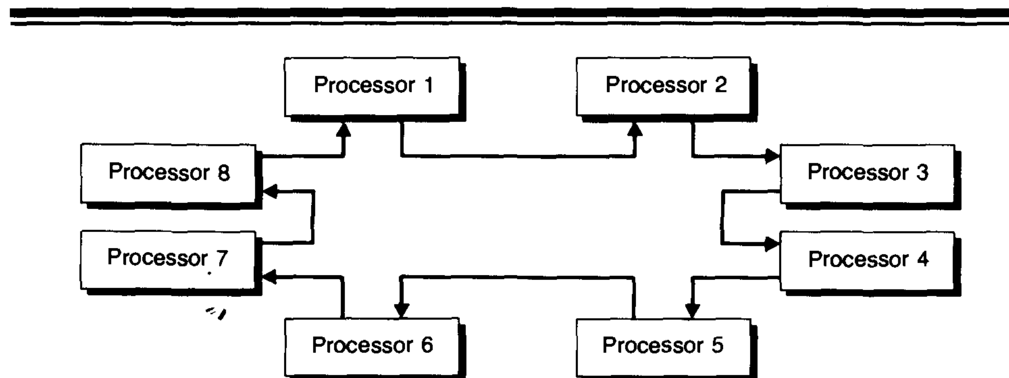


Fig. 6.5 A multiprocessor based on a loop interconnection.

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 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 protocol 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 10 to 50 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 100 MHz. However, if a bus leaves a chip, then maximum clock rates fall back to the 10-to-50 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 a clock speed of 400 MHz is readily achievable. Clock rates exceeding 1 GHz should eventually be released commercially.

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.

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

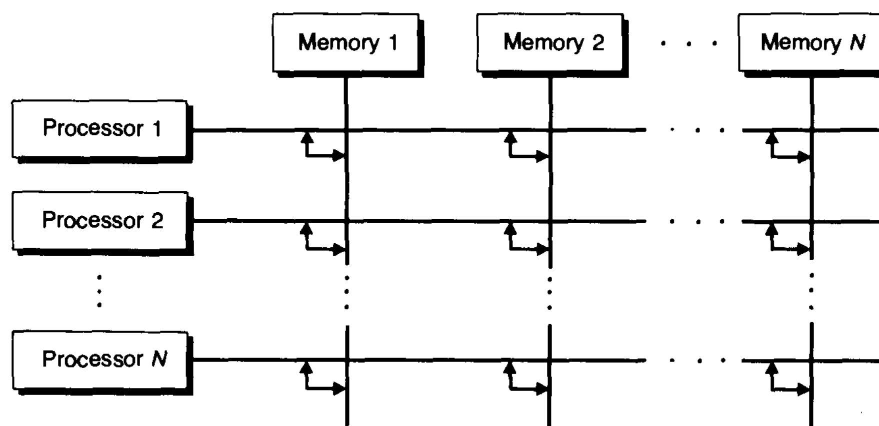


Fig. 6.6 An $N \times N$ crossbar switch in an N-processor multiprocessor. At each crossing in the network is a switch that permits any processor to connect to any memory.

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 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.

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

```
Procedure Add_to_Sum (var Global_Sum: Real, 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);
  Global_Sum := Global_Sum + Local_Sum;
  UNLOCK(Global_Sum);
end {* Procedure Add_to_Sum *};
```

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*) and UNLOCK(*Global_Sum*) that surround the read/modify/write operation on *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. It has to be implemented very carefully in both hardware and software because it is prone to error.

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 the memory. When the bus is replaced by a crossbar, communications bottlenecks disappear, but performance is limited by the next tightest bottleneck, which might 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 16-fold larger memory 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 processes. 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 processes.

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.

If within an application the architect discovers inherent difficulties that limit performance, then another approach is required. The following section describes an implementation technique that offers a unique way to update a shared variable without forcing the update to be executed serially. For some problems, this approach might be the only means available to avoid a shared-memory bottleneck.

6.3.4 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 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, and some applications may not satisfy these conditions. Those applications have a fundamental bottleneck stemming from contention for access to shared variables, and unless another advance in technology becomes available, multiprocessor architectures may be unsuitable for these problems except for small values of N.

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. Moreover, no contention occurs if Processor i requests data from Memory $pi + c$, where p is an odd number, provided that 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, for example, each of the $\log N$

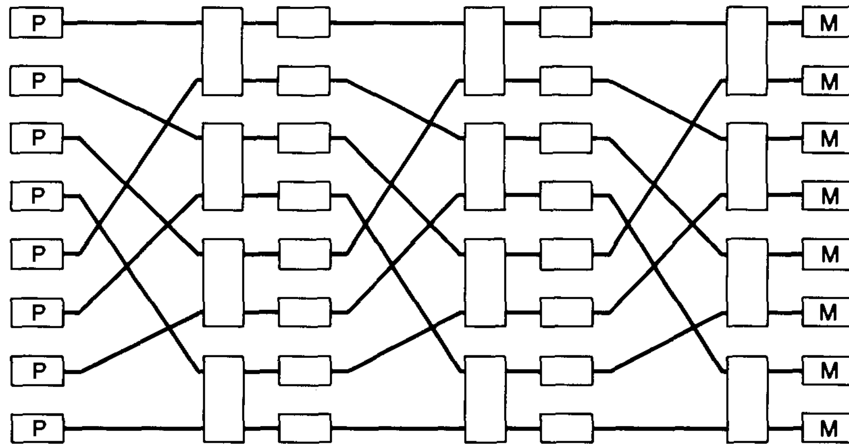


Fig. 6.7 A shuffle-exchange network for connecting eight processors to eight memories. Processors are labeled with *P* and memories with *M*.

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.

6.3.5 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 [1986] 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

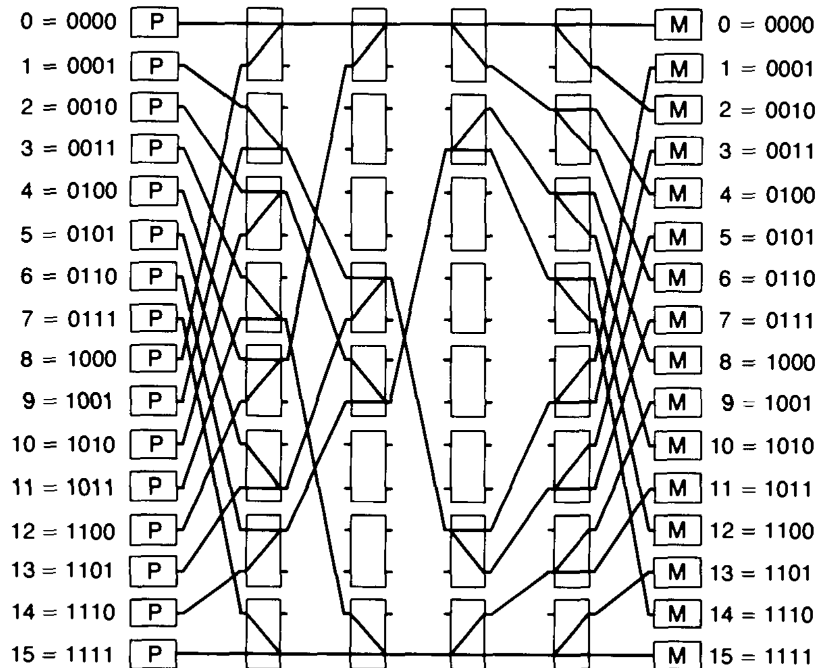


Fig. 6.8 The interconnections used to create a reverse-binary transformation in a shuffle-exchange network. Note that only some of the interconnections are used among the internal paths of the network.

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. This places

the input to the last step in reverse-binary order, rather than lexical order. For such an input, the FFT produces an output that is in lexical order. Hence, no reverse-binary transformation is performed, 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; or
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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 39

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:

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

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:

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

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 one 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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 40

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.

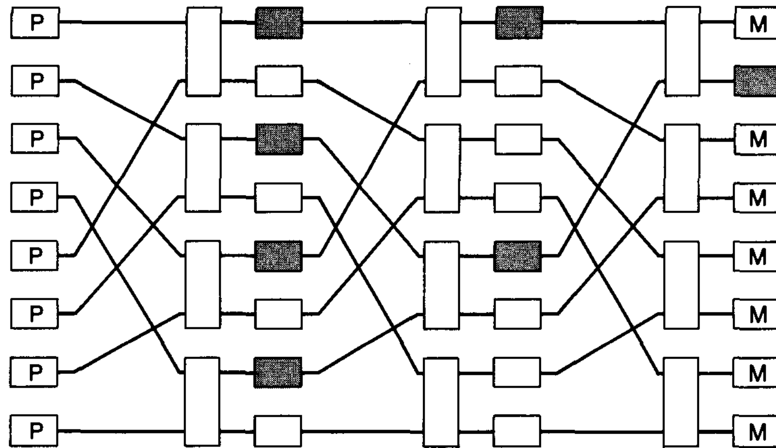


Fig. 6.9 A "hot" spot in a memory module (indicated by shading) and the switching modules that block as a result. The path from Processor 0 (the top processor) to Memory 3 is blocked, although neither Processor 0 nor Memory 3 is very active.

6.3.6 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.

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 requesters 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 requesters is the same as some set of values that would be returned to the M requesters 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 bandwidth 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 requesters. 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

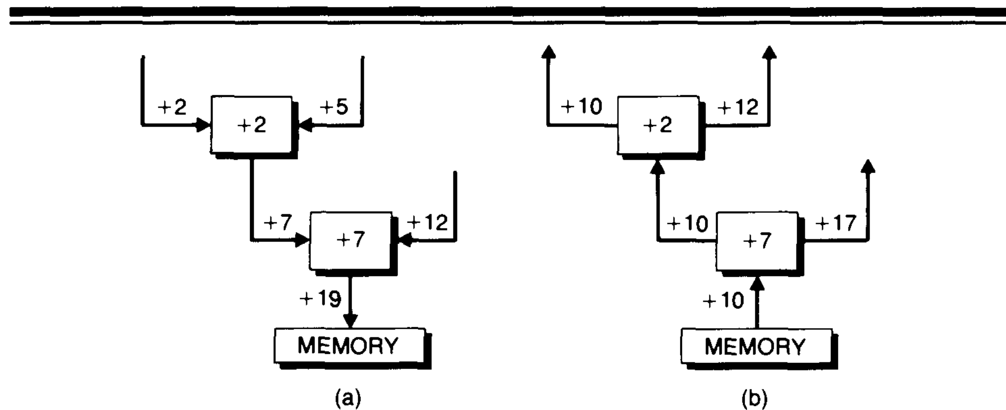


Fig. 6.10 Two phases of a Fetch-and-Add instruction:
(a) The data flow towards memory when increments of 2, 5, and 12 are applied. The numbers in the switch cells show the saved datum; and
(b) The data flow away from memory for the return of information to the requesting processors. The memory returns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

combines data moving towards 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 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 has large gains if it is successful in reducing hot-spot contention by removing critical sections for some shared variables.

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 case depicted here may seem artificial, but it is quite realistic. Programs are often written with barriers at which processors must halt until all processors reach the barrier. At the moment when the last process reaches the barrier, all processors become free. Hence, the normal case at a barrier is for N processors to become free simultaneously. When this occurs, they all reach for new work at the same queue, and the queue becomes a severe bottleneck.

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 processes, 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. It is the only mechanism proposed to date that is seriously being implemented for solving the hot-spot problem and for eliminating serial bottlenecks in multiprocessor code.

Our discussion has mentioned the potential of Fetch-and-Add, but the concept has not been fully evaluated at the time of this writing. Several questions have to be resolved to determine if Fetch-and-Add and the combining network will truly be cost-effective for multiprocessors.

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

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 45

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] propose 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.

This design undoubtedly influenced the inventors of the combining switch, but it is generally less useful than is the combining switch, which eliminates the major bottleneck of a critical section in an enqueue/dequeue routine. Sullivan *et al.* did not solve the serialization problem and this severely restricts the utility of their idea.

Can a combining network actually eliminate hot-spot contention? A hot memory can be hot if it receives a disproportionate number of accesses, but a combining network is effective only if all those accesses are to the same address. Is this case realistic? Perhaps it is if the reason for the biased distribution of accesses is due to accesses to shared data.

A research effort that is exploring this question and many other related ones is the RP3 project at IBM [Pfister *et al.* 1985]. Its structure is outlined in Fig. 6.11. At the left is a processor, one of 512 in the largest configuration planned, and at the right is a combining network comprised of shuffle-exchange stages.

This network 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 pro-

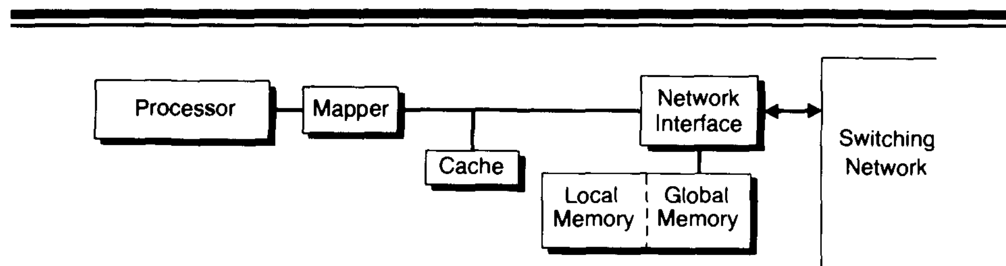


Fig. 6.11 The structure of one of the 512 processors of the full implementation of the IBM RP3. The switching network is a shuffle-exchange network with combining logic.

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 46

cessors 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 extremely advantageous to multiplex 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 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.

In closing this section, we mention that there is a trade-off in time and cost in the selection of interconnections. The shuffle-exchange network lies somewhere in the middle of the possible trade-offs, where buses represent one extreme and crossbars represent the other.

The shuffle-exchange is not the only network in the middle of the range. There can be higher fan-in and fan-out per switch if increasing fan-in and fan-out can be done inexpensively and reduces delay through the network. Several hypercube computers based on this general principle were

introduced in the mid-1980s, the most parallel being the Connection Machine, with 64K processors [Hillis 1986], and the most influential being the Cosmic Cube [Seitz 1985], with 128 processors. Although neither of these machines incorporates combining switches *per se* in its design, the hypercube connection pattern is an extension of the shuffle-exchange connection, and, consequently, the notion of a combining switch for the shuffle-exchange network extends to the hypercube network by analogy.

Hillis and Steele [1986] describe how the Connection Machine implements combining and serialization in $O(\log_2 N)$ time by means of SIMD broadcast instructions. So in spite of being quite different from the RP3 and Ultracomputer, the Connection Machine's hypercube connections support a similar function.

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 and 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 ten percent of the time, and remote fetches occur only on misses, then the number of processors supportable on the interconnection network is ten

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 48

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, but 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.

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.

In all of the studies, the solutions are limited to ensuring cache coherence

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 50

for shared variables. The stale-data problem caused by moving a program away from and then back to the processor has to be solved by other means, such as by flushing the cache when a program is moved away.

Of the many proposals, our discussion picks a single reasonable solution to cache coherence. 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.

Here are the basic operations that must take place to maintain cache coherence:

1. If a READ operation for a shared datum misses in cache, then all caches in the system must be interrogated for a copy of the datum.
2. All WRITE operations to a shared datum, whether they are hits or misses, force all caches in the system to be checked for a copy of the datum. A possible exception to this rule is if the datum is tagged as being the only cached copy of the data in the system, in which case no external broadcast is necessary.

Before discussing alternatives, note that there is a severe performance penalty associated with cache-coherency protocols. The first requirement causes a broadcast operation followed by a cache read in every cache in the system, which tends to increase network contention and reduces available cache bandwidth. Since this operation takes place only on misses to shared data, its frequency should be just a few percent of the reads on any single processor.

As the number of processors increases, however, the load on the communications network and cache traffic quickly approaches saturation. For example, a one-percent miss rate on shared data in each of 100 processors of a multiprocessor generates $100 \times 0.01 = 1$ broadcast request and one cache read per clock cycle. This broadcast will saturate the communications system and the individual caches of all processors.

Potentially greater degradation is caused by the second requirement, which requires a broadcast on every WRITE to a shared datum unless the system is able to tell that the shared datum is not resident in any other cache. 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. 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-coherency protocol generally causes heavy traffic due to frequent broadcasts that progressively move the

datum from one cache to another as it is read and modified 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-coherency 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 coherency, 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. A cache listener responds to a READ by interrogating its own cache and reporting back the data.

If two or more respondents exist, then any respondent can report back data because the data should be identical. Most protocols, however, provide a unique ownership tag that dictates which respondent should deliver the data requested. When a WRITE is received, a listener can respond either by replacing the local value with the broadcast value or by purging the local value. Which of these is preferred depends on such factors as the cache size and the likelihood of accessing a shared variable again in the immediate future.

The basic protocol provides an opportunity to reduce broadcasts on WRITEs if there is a means for tagging an item in cache to indicate that it is the only copy of the item in any cache. If we add such a tag, then WRITE broadcasts need to occur only for cache misses and for cache hits to items that are resident elsewhere as well. But the tag has to be maintained so that its state is an accurate reflection of the state of the caches.

It is clear that if any datum is flagged as exclusive, then at any point that a broadcast for that item is observed, the tag has to be altered. Each of the proposed protocols provides a means for updating that tag. For example, a possible protocol for maintaining the flag is the following:

1. If the item in a cache is exclusive, and a read request for the item is observed on the network, then change the flag to nonexclusive and deliver

a copy of the item to the requester. The requester tags the item as nonexclusive as well.

2. Purge any item in the cache when a write to that address is observed on the network.
3. When writing a datum to the cache, mark the datum as exclusive and notify other processors that the datum has been written. Write the new value of the item in shared memory as well.

A modification of the protocol avoids the purge noted in Item 2 and retains the new value of the item. If this is done, then the tags throughout the system must show nonexclusive ownership, so that Item 3 marks an item as exclusive if no other cache has the item, and otherwise marks the item as nonexclusive.

We cannot easily judge if the modification gives better or worse performance overall because much depends on the likelihood of repeated accesses to shared variables. The modified protocol gives better performance for heavy use of shared variables, whereas the basic protocol gives better performance when the right decision is to purge the variable when the vacancy in the cache can be put to immediate use holding other data.

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

6.5 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. The combining switch is an example of a new technology that could make a substantial difference in the future. We expect other ideas of this type 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. 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.

PATENT OWNER DIRECTSTREAM LLC
EX. 2154, p. 54

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 6.4, 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.