I

Large-Scale Multiprocessors and Scientific Applications

Hennessy and Patterson should move MPPs to Chapter 11.

Jim Gray, Microsoft Research
when asked about the coverage of massively parallel processors (MPPs) for the third edition in 2000

Unfortunately for companies in the MPP business, the third edition had only ten chapters and the MPP business did not grow as anticipated when the first and second edition were written.

I.1 Introduction

The primary application of large-scale multiprocessors is for true parallel programming, as opposed to multiprogramming or transaction-oriented computing where independent tasks are executed in parallel without much interaction. In true parallel computing, a set of tasks execute in a collaborative fashion on one application. The primary target of parallel computing is scientific and technical applications. In contrast, for loosely coupled commercial applications, such as Web servers and most transaction-processing applications, there is little communication among tasks. For such applications, loosely coupled clusters are generally adequate and most cost-effective, since intertask communication is rare.

Because true parallel computing involves cooperating tasks, the nature of communication between those tasks and how such communication is supported in the hardware is of vital importance in determining the performance of the application. The next section of this appendix examines such issues and the characteristics of different communication models.

In comparison to sequential programs, whose performance is largely dictated by the cache behavior and issues related to instruction-level parallelism, parallel programs have several additional characteristics that are important to performance, including the amount of parallelism, the size of parallel tasks, the frequency and nature of intertask communication, and the frequency and nature of synchronization. These aspects are affected both by the underlying nature of the application as well as by the programming style. Section I.3 reviews the important characteristics of several scientific applications to give a flavor of these issues.

As we saw in Chapter 5, synchronization can be quite important in achieving good performance. The larger number of parallel tasks that may need to synchronize makes contention involving synchronization a much more serious problem in large-scale multiprocessors. Section I.4 examines methods of scaling up the synchronization mechanisms of Chapter 5.

Section I.5 explores the detailed performance of shared-memory parallel applications executing on a moderate-scale shared-memory multiprocessor. As we will see, the behavior and performance characteristics are quite a bit more complicated than those in small-scale shared-memory multiprocessors. Section I.6 discusses the general issue of how to examine parallel performance for different sized multiprocessors. Section I.7 explores the implementation challenges of distributed shared-memory cache coherence, the key architectural approach used in moderate-scale multiprocessors. Sections I.7 and I.8 rely on a basic understanding of interconnection networks, and the reader should at least quickly review Appendix F before reading these sections.

Section I.8 explores the design of one of the newest and most exciting large-scale multiprocessors in recent times, Blue Gene. Blue Gene is a cluster-based multiprocessor, but it uses a custom, highly dense node designed specifically for this function, as opposed to the nodes of most earlier cluster multiprocessors that used a node architecture similar to those in a desktop or smaller-scale multiprocessor

node. By using a custom node design, Blue Gene achieves a significant reduction in the cost, physical size, and power consumption of a node. Blue Gene/L, a 64 K-node version, was the world's fastest computer in 2006, as measured by the linear algebra benchmark, Linpack.

I.2   Interprocessor Communication: The Critical Performance Issue

In multiprocessors with larger processor counts, interprocessor communication becomes more expensive, since the distance between processors increases. Furthermore, in truly parallel applications where the threads of the application must communicate, there is usually more communication than in a loosely coupled set of distinct processes or independent transactions, which characterize many commercial server applications. These factors combine to make efficient interprocessor communication one of the most important determinants of parallel performance, especially for the scientific market.

Unfortunately, characterizing the communication needs of an application and the capabilities of an architecture is complex. This section examines the key hardware characteristics that determine communication performance, while the next section looks at application behavior and communication needs.

Three performance metrics are critical in any hardware communication mechanism:

1. *Communication bandwidth*—Ideally, the communication bandwidth is limited by processor, memory, and interconnection bandwidths, rather than by some aspect of the communication mechanism. The interconnection network determines the maximum communication capacity of the system. The bandwidth in or out of a single node, which is often as important as total system bandwidth, is affected both by the architecture within the node and by the communication mechanism. How does the communication mechanism affect the communication bandwidth of a node? When communication occurs, resources within the nodes involved in the communication are tied up or occupied, preventing other outgoing or incoming communication. When this *occupancy* is incurred for each word of a message, it sets an absolute limit on the communication bandwidth. This limit is often lower than what the network or memory system can provide. Occupancy may also have a component that is incurred for each communication event, such as an incoming or outgoing request. In the latter case, the occupancy limits the communication rate, and the impact of the occupancy on overall communication bandwidth depends on the size of the messages.

2. *Communication latency*—Ideally, the latency is as low as possible. As Appendix F explains:

$$\text{Communication latency} = \text{Sender overhead} + \text{Time of flight} + \text{Transmission time} + \text{Receiver overhead}$$

assuming no contention. Time of flight is fixed and transmission time is determined by the interconnection network. The software and hardware overheads in sending and receiving messages are largely determined by the communication mechanism and its implementation. Why is latency crucial? Latency affects both performance and how easy it is to program a multiprocessor. Unless latency is hidden, it directly affects performance either by tying up processor resources or by causing the processor to wait.

Overhead and occupancy are closely related, since many forms of overhead also tie up some part of the node, incurring an occupancy cost, which in turn limits bandwidth. Key features of a communication mechanism may directly affect overhead and occupancy. For example, how is the destination address for a remote communication named, and how is protection implemented? When naming and protection mechanisms are provided by the processor, as in a shared address space, the additional overhead is small. Alternatively, if these mechanisms must be provided by the operating system for each communication, this increases the overhead and occupancy costs of communication, which in turn reduce bandwidth and increase latency.

3. *Communication latency hiding*—How well can the communication mechanism hide latency by overlapping communication with computation or with other communication? Although measuring this is not as simple as measuring the first two metrics, it is an important characteristic that can be quantified by measuring the running time on multiprocessors with the same communication latency but different support for latency hiding. Although hiding latency is certainly a good idea, it poses an additional burden on the software system and ultimately on the programmer. Furthermore, the amount of latency that can be hidden is application dependent. Thus, it is usually best to reduce latency wherever possible.

Each of these performance measures is affected by the characteristics of the communications needed in the application, as we will see in the next section. The size of the data items being communicated is the most obvious characteristic, since it affects both latency and bandwidth directly, as well as affecting the efficacy of different latency-hiding approaches. Similarly, the regularity in the communication patterns affects the cost of naming and protection, and hence the communication overhead. In general, mechanisms that perform well with smaller as well as larger data communication requests, and irregular as well as regular communication patterns, are more flexible and efficient for a wider class of applications. Of course, in considering any communication mechanism, designers must consider cost as well as performance.

Advantages of Different Communication Mechanisms

The two primary means of communicating data in a large-scale multiprocessor are message passing and shared memory. Each of these two primary communication

mechanisms has its advantages. For shared-memory communication, the advantages include

■ Compatibility with the well-understood mechanisms in use in centralized multiprocessors, which all use shared-memory communication. The OpenMP consortium (see *www.openmp.org* for description) has proposed a standardized programming interface for shared-memory multiprocessors. Although message passing also uses a standard, MPI or Message Passing Interface, this standard is not used either in shared-memory multiprocessors or in loosely coupled clusters in use in throughput-oriented environments.

■ Ease of programming when the communication patterns among processors are complex or vary dynamically during execution. Similar advantages simplify compiler design.

■ The ability to develop applications using the familiar shared-memory model, focusing attention only on those accesses that are performance critical.

■ Lower overhead for communication and better use of bandwidth when communicating small items. This arises from the implicit nature of communication and the use of memory mapping to implement protection in hardware, rather than through the I/O system.

■ The ability to use hardware-controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data, both shared and private. As we will see, caching reduces both latency and contention for accessing shared data. This advantage also comes with a disadvantage, which we mention below.

The major advantages for message-passing communication include the following:

■ The hardware can be simpler, especially by comparison with a scalable shared-memory implementation that supports coherent caching of remote data.

■ Communication is explicit, which means it is simpler to understand. In shared-memory models, it can be difficult to know when communication is occurring and when it is not, as well as how costly the communication is.

■ Explicit communication focuses programmer attention on this costly aspect of parallel computation, sometimes leading to improved structure in a multiprocessor program.

■ Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization.

■ It makes it easier to use sender-initiated communication, which may have some advantages in performance.

■ If the communication is less frequent and more structured, it is easier to improve fault tolerance by using a transaction-like structure. Furthermore,

the less tight coupling of nodes and explicit communication make fault isolation simpler.

■ The very largest multiprocessors use a cluster structure, which is inherently based on message passing. Using two different communication models may introduce more complexity than is warranted.

Of course, the desired communication model can be created in software on top of a hardware model that supports either of these mechanisms. Supporting message passing on top of shared memory is considerably easier: Because messages essentially send data from one memory to another, sending a message can be implemented by doing a copy from one portion of the address space to another. The major difficulties arise from dealing with messages that may be misaligned and of arbitrary length in a memory system that is normally oriented toward transferring aligned blocks of data organized as cache blocks. These difficulties can be overcome either with small performance penalties in software or with essentially no penalties, using a small amount of hardware support.

Supporting shared memory efficiently on top of hardware for message passing is much more difficult. Without explicit hardware support for shared memory, all shared-memory references need to involve the operating system to provide address translation and memory protection, as well as to translate memory references into message sends and receives. Loads and stores usually move small amounts of data, so the high overhead of handling these communications in software severely limits the range of applications for which the performance of software-based shared memory is acceptable. For these reasons, it has never been practical to use message passing to implement shared memory for a commercial system.

I.3   Characteristics of Scientific Applications

The primary use of scalable shared-memory multiprocessors is for true parallel programming, as opposed to multiprogramming or transaction-oriented computing. The primary target of parallel computing is scientific and technical applications. Thus, understanding the design issues requires some insight into the behavior of such applications. This section provides such an introduction.

Characteristics of Scientific Applications

Our scientific/technical parallel workload consists of two applications and two computational kernels. The kernels are fast Fourier transformation (FFT) and an LU decomposition, which were chosen because they represent commonly used techniques in a wide variety of applications and have performance characteristics typical of many parallel scientific applications. In addition, the kernels have small code segments whose behavior we can understand and directly track to specific architectural characteristics. Like many scientific applications, I/O is essentially nonexistent in this workload.

The two applications that we use in this appendix are Barnes and Ocean, which represent two important but very different types of parallel computation. We briefly describe each of these applications and kernels and characterize their basic behavior in terms of parallelism and communication. We describe how the problem is decomposed for a distributed shared-memory multiprocessor; certain data decompositions that we describe are not necessary on multiprocessors that have a single, centralized memory.

The FFT Kernel

The FFT is the key kernel in applications that use spectral methods, which arise in fields ranging from signal processing to fluid flow to climate modeling. The FFT application we study here is a one-dimensional version of a parallel algorithm for a complex number FFT. It has a sequential execution time for n data points of $n \log n$. The algorithm uses a high radix (equal to $\sqrt{n}$) that minimizes communication. The measurements shown in this appendix are collected for a million-point input data set.

There are three primary data structures: the input and output arrays of the data being transformed and the roots of unity matrix, which is precomputed and only read during the execution. All arrays are organized as square matrices. The six steps in the algorithm are as follows:

1. Transpose data matrix.
2. Perform 1D FFT on each row of data matrix.
3. Multiply the roots of unity matrix by the data matrix and write the result in the data matrix.
4. Transpose data matrix.
5. Perform 1D FFT on each row of data matrix.
6. Transpose data matrix.

The data matrices and the roots of unity matrix are partitioned among processors in contiguous chunks of rows, so that each processor's partition falls in its own local memory. The first row of the roots of unity matrix is accessed heavily by all processors and is often replicated, as we do, during the first step of the algorithm just shown. The data transposes ensure good locality during the individual FFT steps, which would otherwise access nonlocal data.

The only communication is in the transpose phases, which require all-to-all communication of large amounts of data. Contiguous subcolumns in the rows assigned to a processor are grouped into blocks, which are transposed and placed into the proper location of the destination matrix. Every processor transposes one block locally and sends one block to each of the other processors in the system. Although there is no reuse of individual words in the transpose, with long cache blocks it makes sense to block the transpose to take advantage of the spatial locality afforded by long blocks in the source matrix.

The LU Kernel

LU is an LU factorization of a dense matrix and is representative of many dense linear algebra computations, such as QR factorization, Cholesky factorization, and eigenvalue methods. For a matrix of size $n \times n$ the running time is n^3 and the parallelism is proportional to n^2. Dense LU factorization can be performed efficiently by blocking the algorithm, using the techniques in Chapter 2, which leads to highly efficient cache behavior and low communication. After blocking the algorithm, the dominant computation is a dense matrix multiply that occurs in the innermost loop. The block size is chosen to be small enough to keep the cache miss rate low and large enough to reduce the time spent in the less parallel parts of the computation. Relatively small block sizes (8×8 or 16×16) tend to satisfy both criteria.

Two details are important for reducing interprocessor communication. First, the blocks of the matrix are assigned to processors using a 2D tiling: The $\frac{n}{B} \times \frac{n}{B}$ (where each block is $B \times B$) matrix of blocks is allocated by laying a grid of size $p \times p$ over the matrix of blocks in a cookie-cutter fashion until all the blocks are allocated to a processor. Second, the dense matrix multiplication is performed by the processor that owns the *destination* block. With this blocking and allocation scheme, communication during the reduction is both regular and predictable. For the measurements in this appendix, the input is a 512×512 matrix and a block of 16×16 is used.

A natural way to code the blocked LU factorization of a 2D matrix in a shared address space is to use a 2D array to represent the matrix. Because blocks are allocated in a tiled decomposition, and a block is not contiguous in the address space in a 2D array, it is very difficult to allocate blocks in the local memories of the processors that own them. The solution is to ensure that blocks assigned to a processor are allocated locally and contiguously by using a 4D array (with the first two dimensions specifying the block number in the 2D grid of blocks, and the next two specifying the element in the block).

The Barnes Application

Barnes is an implementation of the Barnes-Hut *n*-body algorithm solving a problem in galaxy evolution. *N-body algorithms* simulate the interaction among a large number of bodies that have forces interacting among them. In this instance, the bodies represent collections of stars and the force is gravity. To reduce the computational time required to model completely all the individual interactions among the bodies, which grow as n^2, *n*-body algorithms take advantage of the fact that the forces drop off with distance. (Gravity, for example, drops off as $1/d^2$, where d is the distance between the two bodies.) The Barnes-Hut algorithm takes advantage of this property by treating a collection of bodies that are "far away" from another body as a single point at the center of mass of the collection and with mass equal to the collection. If the body is far enough from any body in the collection, then the error introduced will be

negligible. The collections are structured in a hierarchical fashion, which can be represented in a tree. This algorithm yields an *n* log *n* running time with parallelism proportional to *n*.

The Barnes-Hut algorithm uses an octree (each node has up to eight children) to represent the eight cubes in a portion of space. Each node then represents the collection of bodies in the subtree rooted at that node, which we call a *cell*. Because the density of space varies and the leaves represent individual bodies, the depth of the tree varies. The tree is traversed once per body to compute the net force acting on that body. The force calculation algorithm for a body starts at the root of the tree. For every node in the tree it visits, the algorithm determines if the center of mass of the cell represented by the subtree rooted at the node is "far enough away" from the body. If so, the entire subtree under that node is approximated by a single point at the center of mass of the cell, and the force that this center of mass exerts on the body is computed. On the other hand, if the center of mass is not far enough away, the cell must be "opened" and each of its subtrees visited. The distance between the body and the cell, together with the error tolerances, determines which cells must be opened. This force calculation phase dominates the execution time. This appendix takes measurements using 16K bodies; the criterion for determining whether a cell needs to be opened is set to the middle of the range typically used in practice.

Obtaining effective parallel performance on Barnes-Hut is challenging because the distribution of bodies is nonuniform and changes over time, making partitioning the work among the processors and maintenance of good locality of reference difficult. We are helped by two properties: (1) the system evolves slowly, and (2) because gravitational forces fall off quickly, with high probability, each cell requires touching a small number of other cells, most of which were used on the last time step. The tree can be partitioned by allocating each processor a subtree. Many of the accesses needed to compute the force on a body in the subtree will be to other bodies in the subtree. Since the amount of work associated with a subtree varies (cells in dense portions of space will need to access more cells), the size of the subtree allocated to a processor is based on some measure of the work it has to do (e.g., how many other cells it needs to visit), rather than just on the number of nodes in the subtree. By partitioning the octree representation, we can obtain good load balance and good locality of reference, while keeping the partitioning cost low. Although this partitioning scheme results in good locality of reference, the resulting data references tend to be for small amounts of data and are unstructured. Thus, this scheme requires an efficient implementation of shared-memory communication.

The Ocean Application

Ocean simulates the influence of eddy and boundary currents on large-scale flow in the ocean. It uses a restricted red-black Gauss-Seidel multigrid technique to solve a set of elliptical partial differential equations. *Red-black Gauss-Seidel* is an iteration technique that colors the points in the grid so as to consistently update

each point based on previous values of the adjacent neighbors. *Multigrid methods* solve finite difference equations by iteration using hierarchical grids. Each grid in the hierarchy has fewer points than the grid below and is an approximation to the lower grid. A finer grid increases accuracy and thus the rate of convergence, while requiring more execution time, since it has more data points. Whether to move up or down in the hierarchy of grids used for the next iteration is determined by the rate of change of the data values. The estimate of the error at every time step is used to decide whether to stay at the same grid, move to a coarser grid, or move to a finer grid. When the iteration converges at the finest level, a solution has been reached. Each iteration has n^2 work for an $n \times n$ grid and the same amount of parallelism.

The arrays representing each grid are dynamically allocated and sized to the particular problem. The entire ocean basin is partitioned into square subgrids (as close as possible) that are allocated in the portion of the address space corresponding to the local memory of the individual processors, which are assigned responsibility for the subgrid. For the measurements in this appendix we use an input that has 130×130 grid points. There are five steps in a time iteration. Since data are exchanged between the steps, all the processors present synchronize at the end of each step before proceeding to the next. Communication occurs when the boundary points of a subgrid are accessed by the adjacent subgrid in nearest-neighbor fashion.

Computation/Communication for the Parallel Programs

A key characteristic in determining the performance of parallel programs is the ratio of computation to communication. If the ratio is high, it means the application has lots of computation for each datum communicated. As we saw in Section I.2, communication is the costly part of parallel computing; therefore, high computation-to-communication ratios are very beneficial. In a parallel processing environment, we are concerned with how the ratio of computation to communication changes as we increase either the number of processors, the size of the problem, or both. Knowing how the ratio changes as we increase the processor count sheds light on how well the application can be sped up. Because we are often interested in running larger problems, it is vital to understand how changing the data set size affects this ratio.

To understand what happens quantitatively to the computation-to-communication ratio as we add processors, consider what happens separately to computation and to communication as we either add processors or increase problem size. Figure I.1 shows that as we add processors, for these applications, the amount of computation per processor falls proportionately and the amount of communication per processor falls more slowly. As we increase the problem size, the computation scales as the $O(\ )$ complexity of the algorithm dictates. Communication scaling is more complex and depends on details of the algorithm; we describe the basic phenomena for each application in the caption of Figure I.1.

Application	Scaling of computation	Scaling of communication	Scaling of computation-to-communication
FFT	$\dfrac{n\log n}{p}$	$\dfrac{n}{p}$	$\log n$
LU	$\dfrac{n}{p}$	$\dfrac{\sqrt{n}}{\sqrt{p}}$	$\dfrac{\sqrt{n}}{\sqrt{p}}$
Barnes	$\dfrac{n\log n}{p}$	approximately $\dfrac{\sqrt{n}(\log n)}{\sqrt{p}}$	approximately $\dfrac{\sqrt{n}}{\sqrt{p}}$
Ocean	$\dfrac{n}{p}$	$\dfrac{\sqrt{n}}{\sqrt{p}}$	$\dfrac{\sqrt{n}}{\sqrt{p}}$

Figure I.1 Scaling of computation, of communication, and of the ratio are critical factors in determining performance on parallel multiprocessors. In this table, p is the increased processor count and n is the increased dataset size. Scaling is on a per-processor basis. The computation scales up with n at the rate given by $O(\ )$ analysis and scales down linearly as p is increased. Communication scaling is more complex. In FFT, all data points must interact, so communication increases with n and decreases with p. In LU and Ocean, communication is proportional to the boundary of a block, so it scales with dataset size at a rate proportional to the side of a square with n points, namely, $\sqrt{n}$; for the same reason communication in these two applications scales inversely to $\sqrt{p}$. Barnes has the most complex scaling properties. Because of the fall-off of interaction between bodies, the basic number of interactions among bodies that require communication scales as $\sqrt{n}$. An additional factor of $\log n$ is needed to maintain the relationships among the bodies. As processor count is increased, communication scales inversely to $\sqrt{p}$.

The overall computation-to-communication ratio is computed from the individual growth rate in computation and communication. In general, this ratio rises slowly with an increase in dataset size and decreases as we add processors. This reminds us that performing a fixed-size problem with more processors leads to increasing inefficiencies because the amount of communication among processors grows. It also tells us how quickly we must scale dataset size as we add processors to keep the fraction of time in communication fixed. The following example illustrates these trade-offs.

Example   Suppose we know that for a given multiprocessor the Ocean application spends 20% of its execution time waiting for communication when run on four processors. Assume that the cost of each communication event is independent of processor count, which is not true in general, since communication costs rise with processor count. How much faster might we expect Ocean to run on a 32-processor machine with the same problem size? What fraction of the execution time is spent on communication in this case? How much larger a problem should we run if we want the fraction of time spent communicating to be the same?

Answer   The computation-to-communication ratio for Ocean is $\sqrt{n}/\sqrt{p}$, so if the problem size is the same, the communication frequency scales by $\sqrt{p}$. This means that communication time increases by $\sqrt{8}$. We can use a variation on Amdahl's law,

recognizing that the computation is decreased but the communication time is increased. If T_4 is the total execution time for four processors, then the execution time for 32 processors is

$$T_{32} = \text{Compute time} + \text{Communication time}$$

$$= \frac{0.8 \times T_4}{8} + (0.2 \times T_4) \times \sqrt{8}$$

$$= 0.1 \times T_4 + 0.57 \times T_4 = 0.67 \times T_4$$

Hence, the speedup is

$$\text{Speedup} = \frac{T_4}{T_{32}} = \frac{T_4}{0.67 \times T_4} = 1.49$$

and the fraction of time spent in communication goes from 20% to 0.57/ 0.67 = 85%.

For the fraction of the communication time to remain the same, we must keep the computation-to-communication ratio the same, so the problem size must scale at the same rate as the processor count. Notice that, because we have changed the problem size, we cannot fairly compare the speedup of the original problem and the scaled problem. We will return to the critical issue of scaling applications for multiprocessors in Section I.6.

I.4 Synchronization: Scaling Up

In this section, we focus first on synchronization performance problems in larger multiprocessors and then on solutions for those problems.

Synchronization Performance Challenges

To understand why the simple spin lock scheme presented in Chapter 5 does not scale well, imagine a large multiprocessor with all processors contending for the same lock. The directory or bus acts as a point of serialization for all the processors, leading to lots of contention, as well as traffic. The following example shows how bad things can be.

Example   Suppose there are 10 processors on a bus and each tries to lock a variable simultaneously. Assume that each bus transaction (read miss or write miss) is 100 clock cycles long. You can ignore the time of the actual read or write of a lock held in the cache, as well as the time the lock is held (they won't matter much!). Determine the number of bus transactions required for all 10 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0. About how long will it take to process the 10 requests? Assume that the bus is

totally fair so that every pending request is serviced before a new request and that the processors are equally fast.

Answer   When i processes are contending for the lock, they perform the following sequence of actions, each of which generates a bus transaction:

> i load linked operations to access the lock
>
> i store conditional operations to try to lock the lock
>
> 1 store (to release the lock)

Thus, for i processes, there are a total of $2i+1$ bus transactions. Note that this assumes that the critical section time is negligible, so that the lock is released before any other processors whose store conditional failed attempt another load linked.

   Thus, for n processes, the total number of bus operations is

$$\sum_{i=1}^{n} (2i + 1) = n(n + 1) + n = n^2 + 2n$$

For 10 processes there are 120 bus transactions requiring 12,000 clock cycles or 120 clock cycles per lock acquisition!

The difficulty in this example arises from contention for the lock and serialization of lock access, as well as the latency of the bus access. (The fairness property of the bus actually makes things worse, since it delays the processor that claims the lock from releasing it; unfortunately, for any bus arbitration scheme some worst-case scenario does exist.) The key advantages of spin locks—that they have low overhead in terms of bus or network cycles and offer good performance when locks are reused by the same processor—are both lost in this example. We will consider alternative implementations in the next section, but before we do that, let's consider the use of spin locks to implement another common high-level synchronization primitive.

Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier*. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one to protect a counter that tallies the processes arriving at the barrier and one to hold the processes until the last process arrives at the barrier. To implement a barrier, we usually use the ability to spin on a variable until it satisfies a test; we use the notation `spin(condition)` to indicate this. Figure I.2 is a typical implementation, assuming that `lock` and `unlock` provide basic spin locks and `total` is the number of processes that must reach the barrier.

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/* first=>reset release */
count = count + 1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) {/* all arrived */
        count=0;/* reset counter */
        release=1;/* release processes */
}
else {/* more to come */
        spin (release==1);/* wait for arrivals */
}
```

Figure I.2 Code for a simple barrier. The lock `counterlock` protects the counter so that it can be atomically incremented. The variable `count` keeps the tally of how many processes have reached the barrier. The variable `release` is used to hold the processes until the last one reaches the barrier. The operation `spin (release==1)` causes a process to wait until all processes reach the barrier.

In practice, another complication makes barrier implementation slightly more complex. Frequently a barrier is used within a loop, so that processes released from the barrier would do some work and then reach the barrier again. Assume that one of the processes never actually leaves the barrier (it stays at the spin operation), which could happen if the OS scheduled another process, for example. Now it is possible that one process races ahead and gets to the barrier again before the last process has left. The "fast" process then traps the remaining "slow" process in the barrier by resetting the flag `release`. Now all the processes will wait infinitely at the next instance of this barrier because one process is trapped at the last instance, and the number of processes can never reach the value of total.

The important observation in this example is that the programmer did nothing wrong. Instead, the implementer of the barrier made some assumptions about forward progress that cannot be assumed. One obvious solution to this is to count the processes as they exit the barrier (just as we did on entry) and not to allow any process to reenter and reinitialize the barrier until all processes have left the prior instance of this barrier. This extra step would significantly increase the latency of the barrier and the contention, which as we will see shortly are already large. An alternative solution is a *sense-reversing barrier*, which makes use of a private per-process variable, `local_sense`, which is initialized to 1 for each process. Figure I.3 shows the code for the sense-reversing barrier. This version of a barrier is safely usable; as the next example shows, however, its performance can still be quite poor.

```
local_sense =! local_sense; /* toggle local_sense */
lock (counterlock);/* ensure update atomic */
count=count+1;/* count arrivals */
if (count==total) {/* all arrived */
      count=0;/* reset counter */
      release=local_sense;/* release processes */
}
unlock (counterlock);/* unlock */
spin (release==local_sense);/* wait for signal */
}
```

Figure I.3   Code for a sense-reversing barrier. The key to making the barrier reusable is the use of an alternating pattern of values for the flag `release`, which controls the exit from the barrier. If a process races ahead to the next instance of this barrier while some other processes are still in the barrier, the fast process cannot trap the other processes, since it does not reset the value of `release` as it did in Figure I.2.

Example   Suppose there are 10 processors on a bus and each tries to execute a barrier simultaneously. Assume that each bus transaction is 100 clock cycles, as before. You can ignore the time of the actual read or write of a lock held in the cache as the time to execute other nonsynchronization operations in the barrier implementation. Determine the number of bus transactions required for all 10 processors to reach the barrier, be released from the barrier, and exit the barrier. Assume that the bus is totally fair, so that every pending request is serviced before a new request and that the processors are equally fast. Don't worry about counting the processors out of the barrier. How long will the entire process take?

Answer   We assume that load linked and store conditional are used to implement lock and unlock. Figure I.4 shows the sequence of bus events for a processor to traverse the barrier, assuming that the first process to grab the bus does not have the lock. There is a slight difference for the last process to reach the barrier, as described in the caption.

For the ith process, the number of bus transactions is $3i+4$. The last process to reach the barrier requires one less. Thus, for n processes, the number of bus transactions is

$$\left(\sum_{i=1}^{n}(3i+4)\right)-1=\frac{3n^2+11n}{2}-1$$

For 10 processes, this is 204 bus cycles or 20,400 clock cycles! Our barrier operation takes almost twice as long as the 10-processor lock-unlock sequence.

Event	Number of times for process *i*	Corresponding source line	Comment
LL `counterlock`	*i*	`lock (counterlock);`	All processes try for lock.
Store conditional	*i*	`lock (counterlock);`	All processes try for lock.
LD `count`	1	`count = count + 1;`	Successful process.
Load linked	*i* − 1	`lock (counterlock);`	Unsuccessful process; try again.
SD `count`	1	`count = count + 1;`	Miss to get exclusive access.
SD `counterlock`	1	`unlock(counterlock);`	Miss to get the lock.
LD `release`	2	`spin (release==local_sense);/`	Read release: misses initially and when finally written.

Figure I.4 **Here are the actions, which require a bus transaction, taken when the *i*th process reaches the barrier.** The last process to reach the barrier requires one less bus transaction, since its read of release for the spin will hit in the cache!

As we can see from these examples, synchronization performance can be a real bottleneck when there is substantial contention among multiple processes. When there is little contention and synchronization operations are infrequent, we are primarily concerned about the latency of a synchronization primitive—that is, how long it takes an individual process to complete a synchronization operation. Our basic spin lock operation can do this in two bus cycles: one to initially read the lock and one to write it. We could improve this to a single bus cycle by a variety of methods. For example, we could simply spin on the swap operation. If the lock were almost always free, this could be better, but if the lock were not free, it would lead to lots of bus traffic, since each attempt to lock the variable would lead to a bus cycle. In practice, the latency of our spin lock is not quite as bad as we have seen in this example, since the write miss for a data item present in the cache is treated as an upgrade and will be cheaper than a true read miss.

The more serious problem in these examples is the serialization of each process's attempt to complete the synchronization. This serialization is a problem when there is contention because it greatly increases the time to complete the synchronization operation. For example, if the time to complete all 10 lock and unlock operations depended only on the latency in the uncontended case, then it would take 1000 rather than 15,000 cycles to complete the synchronization operations. The barrier situation is as bad, and in some ways worse, since it is highly likely to incur contention. The use of a bus interconnect exacerbates these problems, but serialization could be just as serious in a directory-based multiprocessor, where the latency would be large. The next subsection presents some solutions that are useful when either the contention is high or the processor count is large.

Synchronization Mechanisms for Larger-Scale Multiprocessors

What we would like are synchronization mechanisms that have low latency in uncontended cases and that minimize serialization in the case where contention is significant. We begin by showing how software implementations can improve the performance of locks and barriers when contention is high; we then explore two basic hardware primitives that reduce serialization while keeping latency low.

Software Implementations

The major difficulty with our spin lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails. Figure I.5 shows how a spin lock with *exponential back-off* is implemented. Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses (see Sections F.4 to F.8). This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer

```
            DADDUI  R3,R0,#1   ;R3 = initial delay
    lockit: LL      R2,0(R1)   ;load linked
            BNEZ    R2,lockit  ;not available-spin
            DADDUI  R2,R2,#1   ;get locked value
            SC      R2,0(R1)   ;store conditional
            BNEZ    R2,gotit   ;branch if store succeeds
            DSLL    R3,R3,#1   ;increase delay by factor of 2
            PAUSE   R3         ;delays by value in R3
            J       lockit
     gotit: use data protected by lock
```

Figure I.5   A spin lock with exponential back-off. When the store conditional fails, the process delays itself by the value in R3. The delay can be implemented by decrementing a copy of the value in R3 until it reaches 0. The exact timing of the delay is multiprocessor dependent, although it should start with a value that is approximately the time to perform the critical section and release the lock. The statement pause R3 should cause a delay of R3 of these time units. The value in R3 is increased by a factor of 2 every time the store conditional fails, which causes the process to wait twice as long before trying to acquire the lock again. The small variations in the rate at which competing processors execute instructions are usually sufficient to ensure that processes will not continually collide. If the natural perturbation in execution time was insufficient, R3 could be initialized with a small random value, increasing the variance in the successive delays and reducing the probability of successive collisions.

performance when the lock was in use by only two processes and the first one happened to find it locked.

Another technique for implementing locks is to use queuing locks. Queuing locks work by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access. This eliminates contention for a lock when it is freed. We show how queuing locks operate in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits. Before we look at hardware primitives, let's look at a better mechanism for barriers.

Our barrier implementation suffers from contention both during the *gather* stage, when we must atomically update the count, and at the *release* stage, when all the processes must read the release flag. The former is more serious because it requires exclusive access to the synchronization variable and thus creates much more serialization; in comparison, the latter generates only read contention. We can reduce the contention by using a *combining tree*, a structure where multiple requests are locally combined in tree fashion. The same combining tree can be used to implement the release process, reducing the contention there.

Our combining tree barrier uses a predetermined *n*-ary tree structure. We use the variable k to stand for the fan-in; in practice, $k = 4$ seems to work well. When the kth process arrives at a node in the tree, we signal the next level in the tree. When a process arrives at the root, we release all waiting processes. As in our earlier example, we use a sense-reversing technique. A tree-based barrier, as shown in Figure I.6, uses a tree to combine the processes and a single signal to release the barrier. Some MPPs (e.g., the T3D and CM-5) have also included hardware support for barriers, but more recent machines have relied on software libraries for this support.

Hardware Primitives

In this subsection, we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases, we can create a hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This sequence happens on each of the 10 lock/unlock sequences, as we saw in the example on page I-12.

We can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock*. Queuing locks can be implemented either in hardware, which we

```
struct node{/* a node in the combining tree */
      int counterlock; /* lock for this node */
      int count; /* counter for this node */
      int parent; /* parent in the tree = 0..P-1 except for root */
};
struct node tree [0..P-1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release; /* global release flag */

/* function to implement barrier */
barrier (int mynode, int local_sense) {
      lock (tree[mynode].counterlock); /* protect count */
      tree[mynode].count=tree[mynode].count+1;
            /* increment count */
      if (tree[mynode].count==k) {/* all arrived at mynode */
            if (tree[mynode].parent >=0) {
                  barrier(tree[mynode].parent);
            } else{
                  release = local_sense;
            };
            tree[mynode].count = 0; /* reset for the next time */
      unlock (tree[mynode].counterlock); /* unlock */
      spin (release==local_sense); /* wait */
};
/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);
```

Figure I.6 An implementation of a tree-based barrier reduces contention considerably. The tree is assumed to be prebuilt statically using the nodes in the array tree. Each node in the tree combines *k* processes and provides a separate counter and lock, so that at most *k* processes contend at each node. When the *k*th process reaches a node in the tree, it goes up to the parent, incrementing the count at the parent. When the count in the parent node reaches *k*, the release flag is set. The count in each node is reset by the last process to arrive. Sense-reversing is used to avoid races as in the simple barrier. The value of `tree[root].parent` should be set to −1 when the tree is initially built.

describe here, or in software using an array to keep track of the waiting processes. The basic concepts are the same in either case. Our hardware implementation assumes a directory-based multiprocessor where the individual processor caches are addressable. In a bus-based multiprocessor, a software implementation would be more appropriate and would have each processor using a different address for the lock, permitting the explicit transfer of the lock from one process to another.

How does a queuing lock work? On the first miss to the lock variable, the miss is sent to a synchronization controller, which may be integrated with the memory controller (in a bus-based system) or with the directory controller. If the lock is free, it is simply returned to the processor. If the lock is unavailable,

the controller creates a record of the node's request (such as a bit in a vector) and sends the processor back a locked value for the variable, which the processor then spins on. When the lock is freed, the controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processor's cache or invalidate the copy, causing the processor to miss and fetch an available copy of the lock.

Example   How many bus transactions and how long does it take to have 10 processors lock and unlock the variable using a queuing lock that updates the lock on a miss? Make the other assumptions about the system the same as those in the earlier example on page I-12.

Answer   For n processors, each will initially attempt a lock access, generating a bus transaction; one will succeed and free up the lock, for a total of $n+1$ transactions for the first processor. Each subsequent processor requires two bus transactions, one to receive the lock and one to free it up. Thus, the total number of bus transactions is $(n+1)+2(n-1)=3n-1$. Note that the number of bus transactions is now linear in the number of processors contending for the lock, rather than quadratic, as it was with the spin lock we examined earlier. For 10 processors, this requires 29 bus cycles or 2900 clock cycles.

There are a couple of key insights in implementing such a queuing lock capability. First, we need to be able to distinguish the initial access to the lock, so we can perform the queuing operation, and also the lock release, so we can provide the lock to another processor. The queue of waiting processes can be implemented by a variety of mechanisms. In a directory-based multiprocessor, this queue is akin to the sharing set, and similar hardware can be used to implement the directory and queuing lock operations. One complication is that the hardware must be prepared to reclaim such locks, since the process that requested the lock may have been context-switched and may not even be scheduled again on the same processor.

Queuing locks can be used to improve the performance of our barrier operation. Alternatively, we can introduce a primitive that reduces the amount of time needed to increment the barrier count, thus reducing the serialization at this bottleneck, which should yield comparable performance to using queuing locks. One primitive that has been introduced for this and for building other synchronization operations is *fetch-and-increment*, which atomically fetches a variable and increments its value. The returned value can be either the incremented value or the fetched value. Using fetch-and-increment we can dramatically improve our barrier implementation, compared to the simple code-sensing barrier.

Example   Write the code for the barrier using fetch-and-increment. Making the same assumptions as in our earlier example and also assuming that a fetch-and-increment operation, which returns the incremented value, takes 100 clock cycles, determine the time for 10 processors to traverse the barrier. How many bus cycles are required?

```
local_sense =! local_sense; /* toggle local_sense */
fetch_and_increment(count);/* atomic update */
if (count==total) {/* all arrived */
      count=0;/* reset counter */
      release=local_sense;/* release processes */
}
else {/* more to come */
      spin (release==local_sense);/* wait for signal */
}
```

Figure I.7 Code for a sense-reversing barrier using fetch-and-increment to do the counting.

Answer   Figure I.7 shows the code for the barrier. For *n* processors, this implementation requires *n* fetch-and-increment operations, *n* cache misses to access the count, and *n* cache misses for the release operation, for a total of 3*n* bus transactions. For 10 processors, this is 30 bus transactions or 3000 clock cycles. Like the queueing lock, the time is linear in the number of processors. Of course, fetch-and-increment can also be used in implementing the combining tree barrier, reducing the serialization at each node in the tree.

As we have seen, synchronization problems can become quite acute in largerscale multiprocessors. When the challenges posed by synchronization are combined with the challenges posed by long memory latency and potential load imbalance in computations, we can see why getting efficient usage of large-scale parallel processors is very challenging.

I.5   Performance of Scientific Applications on Shared-Memory Multiprocessors

This section covers the performance of the scientific applications from Section I.3 on both symmetric shared-memory and distributed shared-memory multiprocessors.

Performance of a Scientific Workload on a Symmetric Shared-Memory Multiprocessor

We evaluate the performance of our four scientific applications on a symmetric shared-memory multiprocessor using the following problem sizes:

■   *Barnes-Hut*—16 K bodies run for six time steps (the accuracy control is set to 1.0, a typical, realistic value)

■ *FFT*—1 million complex data points

■ *LU*—A 512×512 matrix is used with 16×16 blocks

■ *Ocean*—A 130×130 grid with a typical error tolerance

In looking at the miss rates as we vary processor count, cache size, and block size, we decompose the total miss rate into *coherence misses* and normal uniprocessor misses. The normal uniprocessor misses consist of capacity, conflict, and compulsory misses. We label these misses as capacity misses because that is the dominant cause for these benchmarks. For these measurements, we include as a coherence miss any write misses needed to upgrade a block from shared to exclusive, even though no one is sharing the cache block. This measurement reflects a protocol that does not distinguish between a private and shared cache block.

Figure I.8 shows the data miss rates for our four applications, as we increase the number of processors from 1 to 16, while keeping the problem size constant. As we increase the number of processors, the total amount of cache increases, usually causing the capacity misses to drop. In contrast, increasing the processor count usually causes the amount of communication to increase, in turn causing the coherence misses to rise. The magnitude of these two effects differs by application.

In FFT, the capacity miss rate drops (from nearly 7% to just over 5%) but the coherence miss rate increases (from about 1% to about 2.7%), leading to a constant overall miss rate. Ocean shows a combination of effects, including some that relate to the partitioning of the grid and how grid boundaries map to cache blocks. For a typical 2D grid code the communication-generated misses are proportional to the boundary of each partition of the grid, while the capacity misses are proportional to the area of the grid. Therefore, increasing the total amount of cache while keeping the total problem size fixed will have a more significant effect on the capacity miss rate, at least until each subgrid fits within an individual processor's cache. The significant jump in miss rate between one and two processors occurs because of conflicts that arise from the way in which the multiple grids are mapped to the caches. This conflict is present for direct-mapped and two-way set associative caches, but fades at higher associativities. Such conflicts are not unusual in array-based applications, especially when there are multiple grids in use at once. In Barnes and LU, the increase in processor count has little effect on the miss rate, sometimes causing a slight increase and sometimes causing a slight decrease.

Increasing the cache size usually has a beneficial effect on performance, since it reduces the frequency of costly cache misses. Figure I.9 illustrates the change in miss rate as cache size is increased for 16 processors, showing the portion of the miss rate due to coherence misses and to uniprocessor capacity misses. Two effects can lead to a miss rate that does not decrease—at least not as quickly as we might expect—as cache size increases: inherent communication and plateaus in the miss rate. Inherent communication leads to a certain frequency of coherence misses that are not significantly affected by increasing cache size. Thus, if the cache size is increased while maintaining a fixed problem size, the coherence miss rate
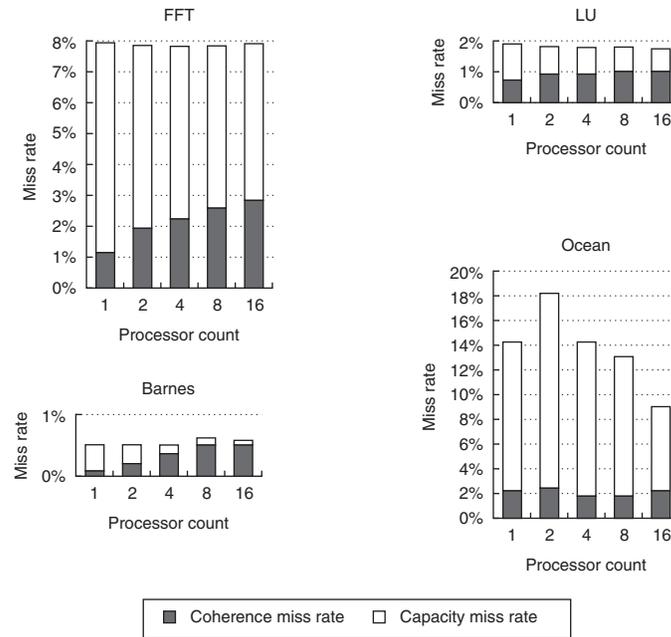
Figure I.8 Data miss rates can vary in nonobvious ways as the processor count is increased from 1 to 16. The miss rates include both coherence and capacity miss rates. The compulsory misses in these benchmarks are all very small and are included in the capacity misses. Most of the misses in these applications are generated by accesses to data that are potentially shared, although in the applications with larger miss rates (FFT and Ocean), it is the capacity misses rather than the coherence misses that comprise the majority of the miss rate. Data are potentially shared if they are allocated in a portion of the address space used for shared data. In all except Ocean, the potentially shared data are heavily shared, while in Ocean only the boundaries of the subgrids are actually shared, although the entire grid is treated as a potentially shared data object. Of course, since the boundaries change as we increase the processor count (for a fixed-size problem), different amounts of the grid become shared. The anomalous increase in capacity miss rate for Ocean in moving from 1 to 2 processors arises because of conflict misses in accessing the subgrids. In all cases except Ocean, the fraction of the cache misses caused by coherence transactions rises when a fixed-size problem is run on an increasing number of processors. In Ocean, the coherence misses initially fall as we add processors due to a large number of misses that are write ownership misses to data that are potentially, but not actually, shared. As the subgrids begin to fit in the aggregate cache (around 16 processors), this effect lessens. The single-processor numbers include write upgrade misses, which occur in this protocol even if the data are not actually shared, since they are in the shared state. For all these runs, the cache size is 64 KB, two-way set associative, with 32-byte blocks. Notice that the scale on the *y*-axis for each benchmark is different, so that the behavior of the individual benchmarks can be seen clearly.
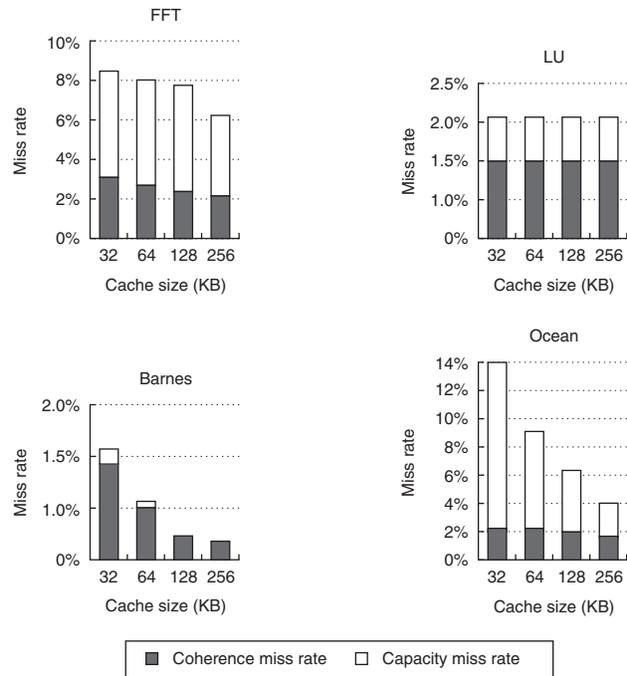
Figure I.9 The miss rate usually drops as the cache size is increased, although coherence misses dampen the effect. The block size is 32 bytes and the cache is two-way set associative. The processor count is fixed at 16 processors. Observe that the scale for each graph is different.

eventually limits the decrease in cache miss rate. This effect is most obvious in Barnes, where the coherence miss rate essentially becomes the entire miss rate.

A less important effect is a temporary plateau in the capacity miss rate that arises when the application has some fraction of its data present in cache but some significant portion of the dataset does not fit in the cache or in caches that are slightly bigger. In LU, a very small cache (about 4 KB) can capture the pair of 16×16 blocks used in the inner loop; beyond that, the next big improvement in capacity miss rate occurs when both matrices fit in the caches, which occurs when the total cache size is between 4 MB and 8 MB. This effect, sometimes called a *working set effect*, is partly at work between 32 KB and 128 KB for FFT, where the capacity miss rate drops only 0.3%. Beyond that cache size, a faster decrease in the capacity miss rate is seen, as a major data structure begins to reside in the cache. These plateaus are common in programs that deal with large arrays in a structured fashion.

Increasing the block size is another way to change the miss rate in a cache. In uniprocessors, larger block sizes are often optimal with larger caches. In

multiprocessors, two new effects come into play: a reduction in spatial locality for shared data and a potential increase in miss rate due to false sharing. Several studies have shown that shared data have lower spatial locality than unshared data. Poorer locality means that, for shared data, fetching larger blocks is less effective than in a uniprocessor because the probability is higher that the block will be replaced before all its contents are referenced. Likewise, increasing the basic size also increases the potential frequency of false sharing, increasing the miss rate.

Figure I.10 shows the miss rates as the cache block size is increased for a 16-processor run with a 64 KB cache. The most interesting behavior is in Barnes, where the miss rate initially declines and then rises due to an increase in the number of coherence misses, which probably occurs because of false sharing. In the other benchmarks, increasing the block size decreases the overall miss rate. In Ocean and LU, the block size increase affects both the coherence and capacity miss rates about equally. In FFT, the coherence miss rate is actually decreased at a faster rate than the capacity miss rate. This reduction occurs because the communication in FFT is structured to be very efficient. In less optimized programs, we would expect more
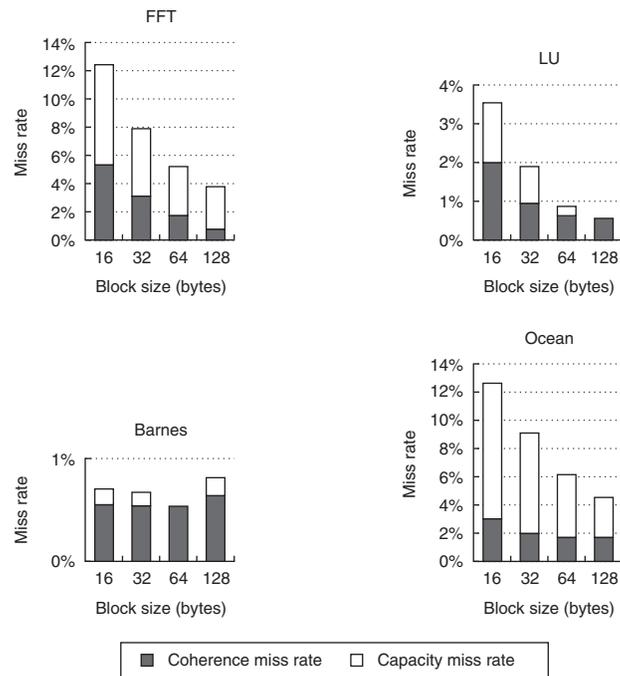
Figure I.10 The data miss rate drops as the cache block size is increased. All these results are for a 16-processor run with a 64 KB cache and two-way set associativity. Once again we use different scales for each benchmark.
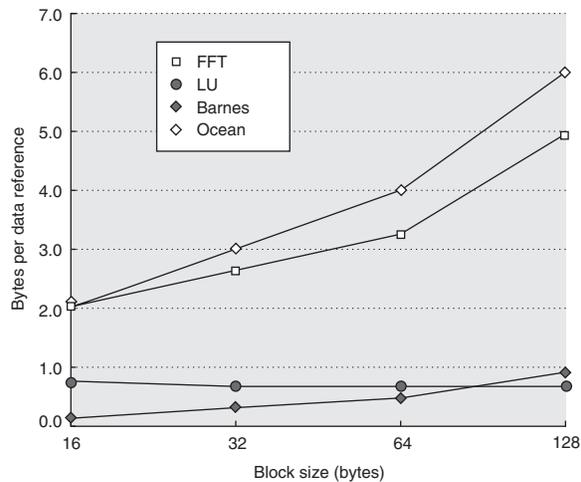
Figure I.11 Bus traffic for data misses climbs steadily as the block size in the data cache is increased. The factor of 3 increase in traffic for Ocean is the best argument against larger block sizes. Remember that our protocol treats ownership or upgrade misses the same as other misses, slightly increasing the penalty for large cache blocks; in both Ocean and FFT, this simplification accounts for less than 10% of the traffic.

false sharing and less spatial locality for shared data, resulting in more behavior like that of Barnes.

Although the drop in miss rates with longer blocks may lead you to believe that choosing a longer block size is the best decision, the bottleneck in bus-based multiprocessors is often the limited memory and bus bandwidth. Larger blocks mean more bytes on the bus per miss. Figure I.11 shows the growth in bus traffic as the block size is increased. This growth is most serious in the programs that have a high miss rate, especially Ocean. The growth in traffic can actually lead to performance slowdowns due both to longer miss penalties and to increased bus contention.

Performance of a Scientific Workload on a Distributed-Memory Multiprocessor

The performance of a directory-based multiprocessor depends on many of the same factors that influence the performance of bus-based multiprocessors (e.g., cache size, processor count, and block size), as well as the distribution of misses to various locations in the memory hierarchy. The location of a requested data item depends on both the initial allocation and the sharing patterns. We start by examining the basic cache performance of our scientific/technical workload and then look at the effect of different types of misses.

Because the multiprocessor is larger and has longer latencies than our snooping-based multiprocessor, we begin with a slightly larger cache (128 KB) and a larger block size of 64 bytes.

In distributed-memory architectures, the distribution of memory requests between local and remote is key to performance because it affects both the consumption of global bandwidth and the latency seen by requests. Therefore, for the figures in this section, we separate the cache misses into local and remote requests. In looking at the figures, keep in mind that, for these applications, most of the remote misses that arise are coherence misses, although some capacity misses can also be remote, and in some applications with poor data distribution such misses can be significant.

As Figure I.12 shows, the miss rates with these cache sizes are not affected much by changes in processor count, with the exception of Ocean, where the miss rate rises at 64 processors. This rise results from two factors: an increase in mapping conflicts in the cache that occur when the grid becomes small, which leads to a rise in local misses, and an increase in the number of the coherence misses, which are all remote.

Figure I.13 shows how the miss rates change as the cache size is increased, assuming a 64-processor execution and 64-byte blocks. These miss rates decrease at rates that we might expect, although the dampening effect caused by little or no reduction in coherence misses leads to a slower decrease in the remote misses than in the local misses. By the time we reach the largest cache size shown, 512 KB, the remote miss rate is equal to or greater than the local miss rate. Larger caches would amplify this trend.

We examine the effect of changing the block size in Figure I.14. Because these applications have good spatial locality, increases in block size reduce the miss rate, even for large blocks, although the performance benefits for going to the largest blocks are small. Furthermore, most of the improvement in miss rate comes from a reduction in the local misses.

Rather than plot the memory traffic, Figure I.15 plots the number of bytes required per data reference versus block size, breaking the requirement into local and global bandwidth. In the case of a bus, we can simply aggregate the demands of each processor to find the total demand for bus and memory bandwidth. For a scalable interconnect, we can use the data in Figure I.15 to compute the required per-node global bandwidth and the estimated bisection bandwidth, as the next example shows.

Example   Assume a 64-processor multiprocessor with 1 GHz processors that sustain one memory reference per processor clock. For a 64-byte block size, the remote miss rate is 0.7%. Find the per-node and estimated bisection bandwidth for FFT. Assume that the processor does not stall for remote memory requests; this might be true if, for example, all remote data were prefetched. How do these bandwidth requirements compare to various interconnection technologies?
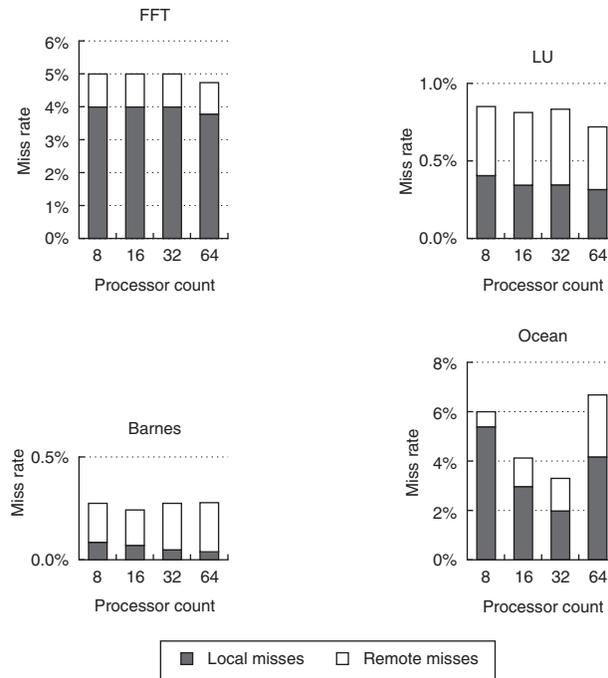
Figure I.12 The data miss rate is often steady as processors are added for these benchmarks. Because of its grid structure, Ocean has an initially decreasing miss rate, which rises when there are 64 processors. For Ocean, the local miss rate drops from 5% at 8 processors to 2% at 32, before rising to 4% at 64. The remote miss rate in Ocean, driven primarily by communication, rises monotonically from 1% to 2.5%. Note that, to show the detailed behavior of each benchmark, different scales are used on the *y*-axis. The cache for all these runs is 128 KB, two-way set associative, with 64-byte blocks. Remote misses include any misses that require communication with another node, whether to fetch the data or to deliver an invalidate. In particular, in this figure and other data in this section, the measurement of remote misses includes write upgrade misses where the data are up to date in the local memory but cached elsewhere and, therefore, require invalidations to be sent. Such invalidations do indeed generate remote traffic, but may or may not delay the write, depending on the consistency model.

FFT performs all-to-all communication, so the bisection bandwidth is equal to the number of processors times the per-node bandwidth, or about 64×448 MB/sec $= 28.7$ GB/sec. The SGI Origin 3000 with 64 processors has a bisection bandwidth of about 50 GB/sec. No standard networking technology comes close.

Answer   The per-node bandwidth is simply the number of data bytes per reference times the reference rate: $0.7\% \times 1$ GB/sec $\times 64 = 448$ MB/sec. This rate is somewhat higher than the hardware sustainable transfer rate for the CrayT3E (using a block prefetch)
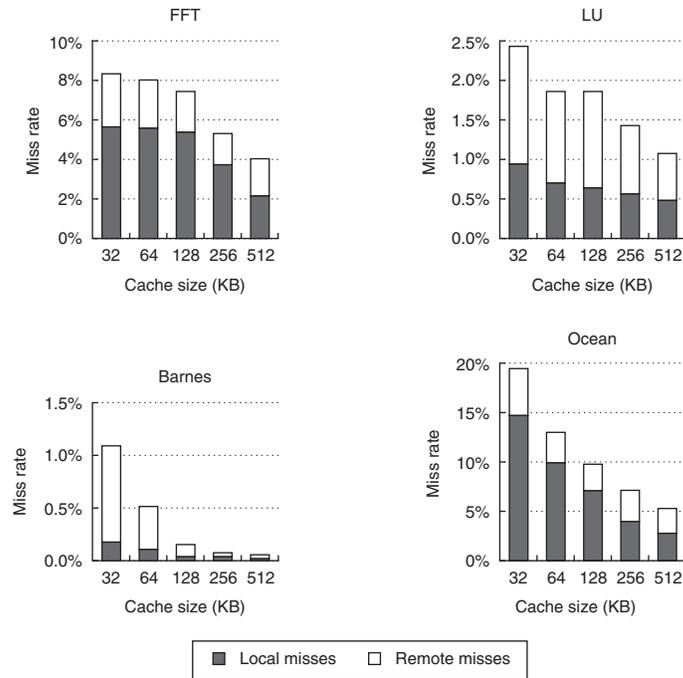
Figure I.13 Miss rates decrease as cache sizes grow. Steady decreases are seen in the local miss rate, while the remote miss rate declines to varying degrees, depending on whether the remote miss rate had a large capacity component or was driven primarily by communication misses. In all cases, the decrease in the local miss rate is larger than the decrease in the remote miss rate. The plateau in the miss rate of FFT, which we mentioned in the last section, ends once the cache exceeds 128 KB. These runs were done with 64 processors and 64-byte cache blocks.

and lower than that for an SGI Origin 3000 (1.6 GB/processor pair). The FFT per-node bandwidth demand exceeds the bandwidth sustainable from the fastest standard networks by more than a factor of 5.

The previous example looked at the bandwidth demands. The other key issue for a parallel program is remote memory access time, or latency. To get insight into this, we use a simple example of a directory-based multiprocessor. Figure I.16 shows the parameters we assume for our simple multiprocessor model. It assumes that the time to first word for a local memory access is 85 processor cycles and that the path to local memory is 16 bytes wide, while the network interconnect is 4 bytes wide. This model ignores the effects of contention, which are probably not too serious in the parallel benchmarks we examine, with the possible exception of FFT, which uses all-to-all communication. Contention could have a serious performance impact in other workloads.
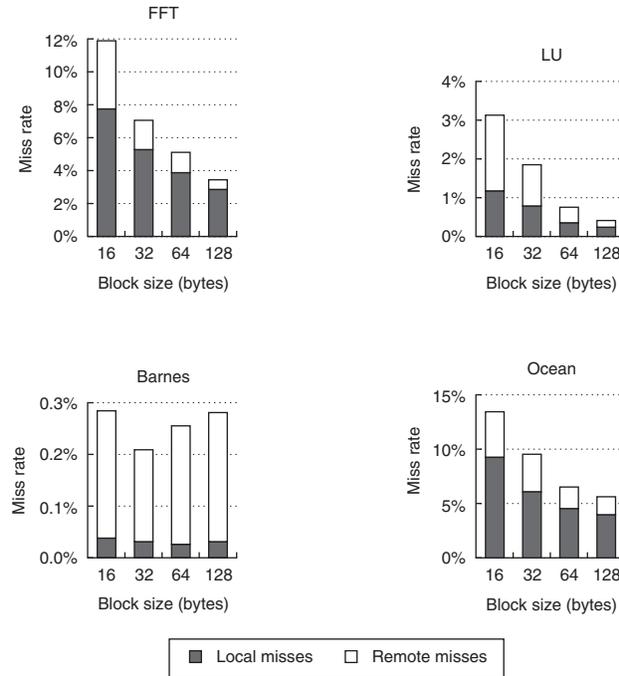
Figure I.14 Data miss rate versus block size assuming a 128 KB cache and 64 processors in total. Although difficult to see, the coherence miss rate in Barnes actually rises for the largest block size, just as in the last section.

Figure I.17 shows the cost in cycles for the average memory reference, assuming the parameters in Figure I.16. Only the latencies for each reference type are counted. Each bar indicates the contribution from cache hits, local misses, remote misses, and three-hop remote misses. The cost is influenced by the total frequency of cache misses and upgrades, as well as by the distribution of the location where the miss is satisfied. The cost for a remote memory reference is fairly steady as the processor count is increased, except for Ocean. The increasing miss rate in Ocean for 64 processors is clear in Figure I.12. As the miss rate increases, we should expect the time spent on memory references to increase also.

Although Figure I.17 shows the memory access cost, which is the dominant multiprocessor cost in these benchmarks, a complete performance model would need to consider the effect of contention in the memory system, as well as the losses arising from synchronization delays.
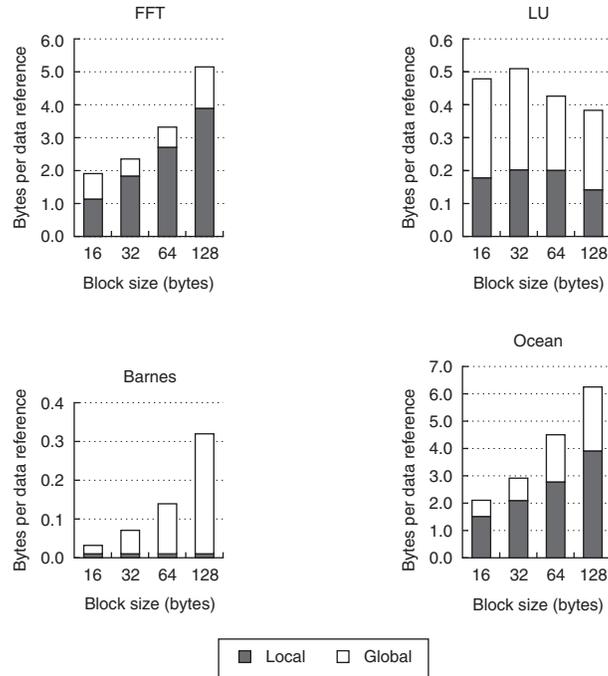
Figure I.15 The number of bytes per data reference climbs steadily as block size is increased. These data can be used to determine the bandwidth required per node both internally and globally. The data assume a 128 KB cache for each of 64 processors.

Characteristic	Processor clock cycles ≤16 processors	Processor clock cycles 17–64 processors
Cache hit	1	1
Cache miss to local memory	85	85
Cache miss to remote home directory	125	150
Cache miss to remotely cached data (three-hop miss)	140	170

Figure I.16 Characteristics of the example directory-based multiprocessor. Misses can be serviced locally (including from the local directory), at a remote home node, or using the services of both the home node and another remote node that is caching an exclusive copy. This last case is called a three-hop miss and has a higher cost because it requires interrogating both the home directory and a remote cache. Note that this simple model does not account for invalidation time but does include some factor for increasing interconnect time. These remote access latencies are based on those in an SGI Origin 3000, the fastest scalable interconnect system in 2001, and assume a 500 MHz processor.
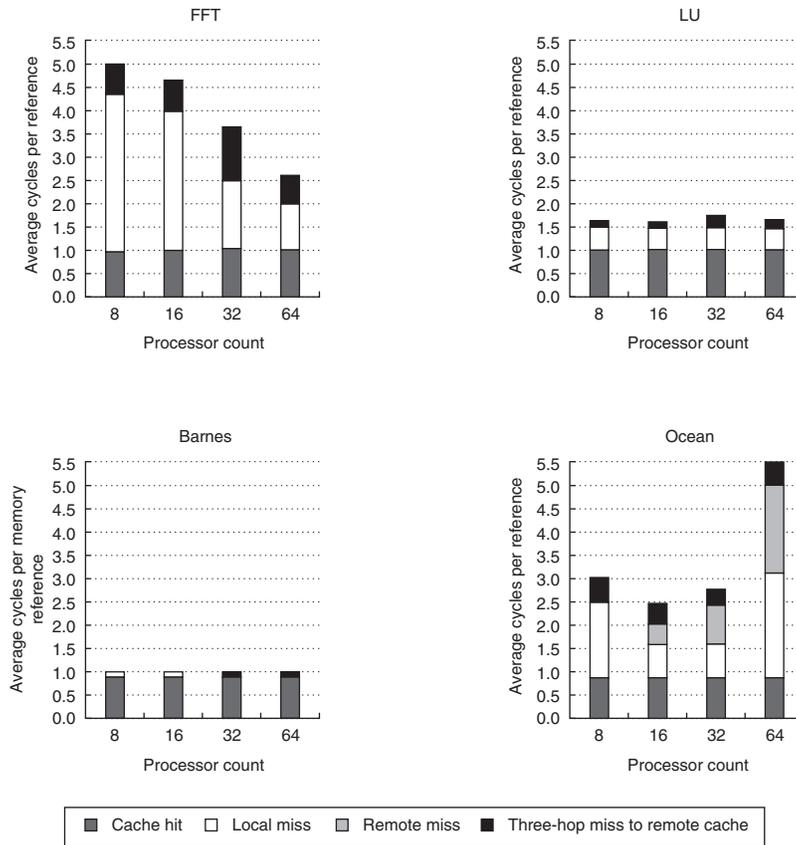
Figure I.17 The effective latency of memory references in a DSM multiprocessor depends both on the relative frequency of cache misses and on the location of the memory where the accesses are served. These plots show the memory access cost (a metric called average memory access time in Chapter 2) for each of the benchmarks for 8, 16, 32, and 64 processors, assuming a 512 KB data cache that is two-way set associative with 64-byte blocks. The average memory access cost is composed of four different types of accesses, with the cost of each type given in Figure I.16. For the Barnes and LU benchmarks, the low miss rates lead to low overall access times. In FFT, the higher access cost is determined by a higher local miss rate (1–4%) and a significant three-hop miss rate (1%). The improvement in FFT comes from the reduction in local miss rate from 4% to 1%, as the aggregate cache increases. Ocean shows the biggest change in the cost of memory accesses, and the highest overall cost at 64 processors. The high cost is driven primarily by a high local miss rate (average 1.6%). The memory access cost drops from 8 to 16 processors as the grids more easily fit in the individual caches. At 64 processors, the dataset size is too small to map properly and both local misses and coherence misses rise, as we saw in Figure I.12.

Performance Measurement of Parallel Processors with Scientific Applications

One of the most controversial issues in parallel processing has been how to measure the performance of parallel processors. Of course, the straightforward answer is to measure a benchmark as supplied and to examine wall-clock time. Measuring wall-clock time obviously makes sense; in a parallel processor, measuring CPU time can be misleading because the processors may be idle but unavailable for other uses.

Users and designers are often interested in knowing not just how well a multiprocessor performs with a certain fixed number of processors, but also how the performance scales as more processors are added. In many cases, it makes sense to scale the application or benchmark, since if the benchmark is unscaled, effects arising from limited parallelism and increases in communication can lead to results that are pessimistic when the expectation is that more processors will be used to solve larger problems. Thus, it is often useful to measure the speedup as processors are added both for a fixed-size problem and for a scaled version of the problem, providing an unscaled and a scaled version of the speedup curves. The choice of how to measure the uniprocessor algorithm is also important to avoid anomalous results, since using the parallel version of the benchmark may understate the uniprocessor performance and thus overstate the speedup.

Once we have decided to measure scaled speedup, the question is *how* to scale the application. Let's assume that we have determined that running a benchmark of size n on p processors makes sense. The question is how to scale the benchmark to run on $m \times p$ processors. There are two obvious ways to scale the problem: (1) keeping the amount of memory used per processor constant, and (2) keeping the total execution time, assuming perfect speedup, constant. The first method, called *memory-constrained scaling*, specifies running a problem of size $m \times n$ on $m \times p$ processors. The second method, called *time-constrained scaling*, requires that we know the relationship between the running time and the problem size, since the former is kept constant. For example, suppose the running time of the application with data size n on p processors is proportional to n^2/p. Then, with time-constrained scaling, the problem to run is the problem whose ideal running time on $m \times p$ processors is still n^2/p. The problem with this ideal running time has size $\sqrt{m} \times n$.

Example   Suppose we have a problem whose execution time for a problem of size n is proportional to n^3. Suppose the actual running time on a 10-processor multiprocessor is 1 hour. Under the time-constrained and memory-constrained scaling models, find the size of the problem to run and the effective running time for a 100-processor multiprocessor.

Answer   For the time-constrained problem, the ideal running time is the same, 1 hour, so the problem size is $\sqrt[3]{10} \times n$ or 2.15 times larger than the original. For

memory-constrained scaling, the size of the problem is $10n$ and the ideal execution time is $10^3/10$, or 100 hours! Since most users will be reluctant to run a problem on an order of magnitude more processors for 100 times longer, this size problem is probably unrealistic.

In addition to the scaling methodology, there are questions as to how the program should be scaled when increasing the problem size affects the quality of the result. Often, we must change other application parameters to deal with this effect. As a simple example, consider the effect of time to convergence for solving a differential equation. This time typically increases as the problem size increases, since, for example, we often require more iterations for the larger problem. Thus, when we increase the problem size, the total running time may scale faster than the basic algorithmic scaling would indicate.

For example, suppose that the number of iterations grows as the log of the problem size. Then, for a problem whose algorithmic running time is linear in the size of the problem, the effective running time actually grows proportional to $n \log n$. If we scaled from a problem of size m on 10 processors, purely algorithmic scaling would allow us to run a problem of size $10\,m$ on 100 processors. Accounting for the increase in iterations means that a problem of size $k \times m$, where $k \log k = 10$, will have the same running time on 100 processors. This problem size yields a scaling of $5.72\,m$, rather than $10\,m$.

In practice, scaling to deal with error requires a good understanding of the application and may involve other factors, such as error tolerances (for example, it affects the cell-opening criteria in Barnes-Hut). In turn, such effects often significantly affect the communication or parallelism properties of the application as well as the choice of problem size.

Scaled speedup is not the same as unscaled (or true) speedup; confusing the two has led to erroneous claims (e.g., see the discussion in Section I.6). Scaled speedup has an important role, but only when the scaling methodology is sound and the results are clearly reported as using a scaled version of the application. Singh, Hennessy, and Gupta [1993] described these issues in detail.

I.7 Implementing Cache Coherence

In this section, we explore the challenge of implementing cache coherence, starting first by dealing with the challenges in a snooping coherence protocol, which we simply alluded to in Chapter 5. Implementing a directory protocol adds some additional complexity to a snooping protocol, primarily arising from the absence of broadcast, which forces the use of a different mechanism to resolve races. Furthermore, the larger processor count of a directory-based multiprocessor means that we cannot retain assumptions of unlimited buffering and must find new ways to avoid deadlock, Let's start with the snooping protocols.

As we mentioned in Chapter 5, the challenge of implementing misses in a snooping coherence protocol without a bus lies in finding a way to make the multi-step miss process appear atomic. Both an upgrade miss and a write miss require the same basic processing and generate the same implementation challenges; for simplicity, we focus on upgrade misses. Here are the steps in handling an upgrade miss:

1. Detect the miss and compose an invalidate message for transmission to other caches.

2. When access to the broadcast communication link is available, transmit the message.

3. When the invalidates have been processed, the processor updates the state of the cache block and then proceeds with the write that caused the upgrade miss.

There are two related difficulties that can arise. First, how will two processors, P1 and P2, that attempt to upgrade the same cache block at the same time resolve the race? Second, when at step 3, how does a processor know when all invalidates have been processed so that it can complete the step?

The solution to finding a winner in the race lies in the ordering imposed by the broadcast communication medium. The communication medium must broadcast any cache miss to all the nodes. If P1 and P2 attempt to broadcast at the same time, we must ensure that either P1's message will reach P2 first or P2's will reach P1 first. This property will be true if there is a single channel through which all ingoing and outgoing requests from a node must pass through and if the communication network does not accept a message unless it can guarantee delivery (i.e., it is effectively circuit switched, see Appendix F). If both P1 and P2 initiate their attempts to broadcast an invalidate simultaneously, then the network can accept only one of these operations and delay the other. This ordering ensures that either P1 or P2 will complete its communication in step 2 first. The network can explicitly signal when it accepts a message and can guarantee it will be the next transmission; alternatively, a processor can simply watch the network for its own request, knowing that once the request is seen, it will be fully transmitted to all processors before any subsequent messages.

Now, suppose P1 wins the race to transmit its invalidate; once it knows it has won the race, it can continue with step 3 and complete the miss handling. There is a potential problem, however, for P2. When P2 undertook step 1, it believed that the block was in the shared state, but for P1 to advance at step 3, it must know that P2 has processed the invalidate, which must change the state of the block at P2 to invalid! One simple solution is for P2 to notice that it has lost the race, by observing that P1's invalidate is broadcast before its own invalidate. P2 can then invalidate the block and generate a write miss to get the data. P1 will see its invalidate before P2's, so it will change the block to modified and update the data, which guarantees forward progress and avoids deadlock. When P1 sees the subsequent invalidate to a block in the Modified state (a possibility that cannot arise in our basic protocol discussed in Chapter 5), it knows that it was the winner of a race. It can simply

ignore the invalidate, knowing that it will be followed by a write miss, or it can write the block back to memory and make its state invalid.

Another solution is to give precedence to incoming requests over outgoing requests, so that before P2 can transmit its invalidate it must handle any pending invalidates or write misses. If any of those misses are for blocks with the same address as a pending outgoing message, the processor must be prepared to restart the write operation, since the incoming request may cause the state of the block to change. Notice that P1 knows that the invalidates will be processed once it has successfully completed the broadcast, since precedence is given to invalidate messages over outgoing requests. (Because it does not employ broadcast, a processor using a directory protocol cannot know when an invalidate is received; instead, explicit acknowledgments are required, as we discuss in the next section. Indeed, as we will see, it cannot even know it has won the race to become the owner until its request is acknowledged.)

Reads will also require a multiple-step process, since we need to get the data back from memory or a remote cache (in a write-back cache system), but reads do not introduce fundamentally new problems beyond what exists for writes.

There are, however, a few additional tricky edge cases that must be handled correctly. For example, in a write-back cache, a processor can generate a read miss that requires a write-back, which it could delay, while giving the read miss priority. If a snoop request appears for the cache block that is to be written back, the processor must discover this and send the data back. Failure to do so can create a deadlock situation. A similar tricky situation exists when a processor generates a write miss, which will make a block exclusive, but, before the processor receives the data and can update the block, other processors generate read misses for that block. The read misses cannot be processed until the writing processor receives the data and updates the block.

One of the more difficult problems occurs in a write-back cache where the data for a read or write miss can come either from memory or from one of the processor caches, but the requesting processor will not know *a priori* where the data will come from. In most bus-based systems, a single global signal is used to indicate whether any processor has the exclusive (and hence the most up-to-date) copy; otherwise, the memory responds. These schemes can work with a pipelined interconnection by requiring that processors signal whether they have the exclusive copy within a fixed number of cycles after the miss is broadcast.

In a modern multiprocessor, however, it is essentially impossible to bound the amount of time required for a snoop request to be processed. Instead, a mechanism is required to determine whether the memory has an up-to-date copy. One solution is to add coherence bits to the memory, indicating whether the data are exclusive in a remote cache. This mechanism begins to move toward the directory approach, whose implementation challenges we consider next.

Implementing Cache Coherence in a DSM Multiprocessor

Implementing a directory-based cache coherence protocol requires overcoming all the problems related to nonatomic actions for a snooping protocol without

the use of broadcast (see Chapter 5), which forced a serialization on competing writes and also ensured the serialization required for the memory consistency model. Avoiding the need to broadcast is a central goal for a directory-based system, so another method for ensuring serialization is necessary.

The serialization of requests for exclusive access to a memory block is easily enforced since those requests will be serialized when they reach the unique directory for the specified block. If the directory controller simply ensures that one request is completely serviced before the next is begun, writes will be serialized. Because the requesters cannot know ahead of time who will win the race and because the communication is not a broadcast, the directory must signal to the winner when it completes the processing of the winner's request. This is done by a message that supplies the data on a write miss or by an explicit acknowledgment message that grants ownership in response to an invalidation request.

What about the loser in this race? The simplest solution is for the system to send a *negative acknowledge*, or *NAK*, which requires that the requesting node regenerate its request. (This is the equivalent of a collision in the broadcast network in a snooping scheme, which requires that one of the transmitting nodes retry its communication.) We will see in the next section why the NAK approach, as opposed to buffering the request, is attractive.

Although the acknowledgment that a requesting node has ownership is completed when the write miss or ownership acknowledgment message is transmitted, we still do not know that the invalidates have been received and processed by the nodes that were in the sharing set. All memory consistency models eventually require (either before the next cache miss or at a synchronization point, for example) that a processor knows that all the invalidates for a write have been processed. In a snooping scheme, the nature of the broadcast network provides this assurance.

How can we know when the invalidates are complete in a directory scheme? The only way to know that the invalidates have been completed is to have the destination nodes of the invalidate messages (the members of the sharing set) explicitly acknowledge the invalidation messages sent from the directory. Who should they be acknowledged to? There are two possibilities. In the first the acknowledgments can be sent to the directory, which can count them, and when all acknowledgments have been received, confirm this with a single message to the original requester. Alternatively, when granting ownership, the directory can tell the register how many acknowledgments to expect. The destinations of the invalidate messages can then send an acknowledgment directly to the requester, whose identity is provided by the directory. Most existing implementations use the latter scheme, since it reduces the possibility of creating a bottleneck at a directory. Although the requirement for acknowledgments is an additional complexity in directory protocols, this requirement arises from the avoidance of a serialization mechanism, such as the snooping broadcast operation, which in itself is the limit to scalability.

Avoiding Deadlock from Limited Buffering

A new complication in the implementation is introduced by the potential scale of a directory-based multiprocessor. In Chapter 5, we assumed that the network could always accept a coherence message and that the request would be acted upon at some point. In a much larger multiprocessor, this assumption of unlimited buffering may be unreasonable. What happens when the network does not have unlimited buffering? The major implication of this limit is that a cache or directory controller may be unable to complete a message send. This could lead to deadlock.

The potential deadlock arises from three properties, which characterize many deadlock situations:

1. More than one resource is needed to complete a transaction: Message buffers are needed to generate requests, create replies and acknowledgments, and accept replies.

2. Resources are held until a nonatomic transaction completes: The buffer used to create the reply cannot be freed until the reply is accepted, for reasons we will see shortly.

3. There is no global partial order on the acquisition of resources: Nodes can generate requests and replies at will.

These characteristics lead to deadlock, and avoiding deadlock requires breaking one of these properties. Freeing up resources without completing a transaction is difficult, since the transaction must be completely backed out and cannot be left half-finished. Hence, our approach will be to try to resolve the need for multiple resources. We cannot simply eliminate this need, but we can try to ensure that the resources will always be available.

One way to ensure that a transaction can always complete is to guarantee that there are always buffers to accept messages. Although this is possible for a small multiprocessor with processors that block on a cache miss or have a small number of outstanding misses, it may not be very practical in a directory protocol, since a single write could generate many invalidate messages. In addition, features such as prefetch and multiple outstanding misses increase the amount of buffering required. There is an alternative strategy, which most systems use and which ensures that a transaction will not actually be initiated until we can guarantee that it has the resources to complete. The strategy has four parts:

1. A separate network (physical or virtual) is used for requests and replies, where a reply is any message that a controller waits for in transitioning between states. This ensures that new requests cannot block replies that will free up buffers.

2. Every request that expects a reply allocates space to accept the reply when the request is generated. If no space is available, the request waits. This ensures that a node can always accept a reply message, which will allow the replying node to free its buffer.

3. Any controller can reject with a NAK any request, but it can never NAK a reply. This prevents a transaction from starting if the controller cannot guarantee that it has buffer space for the reply.

4. Any request that receives a NAK in response is simply retried.

To see that there are no deadlocks with the four properties above, we must ensure that all replies can be accepted and that every request is eventually serviced. Since a cache controller or directory controller always allocates a buffer to handle the reply before issuing a request, it can always accept the reply when it returns. To see that every request is eventually serviced, we need only show that any request could be completed. Since every request starts with a read or write miss at a cache, it is sufficient to show that any read or write miss is eventually serviced. Since the write miss case includes the actions for a read miss as a subset, we focus on showing the write misses are serviced. The simplest situation is when the block is uncached; since that case is subsumed by the case when the block is shared, we focus on the shared and exclusive cases. Let's consider the case where the block is shared:

■ The CPU attempts to do a write and generates a write miss that is sent to the directory. For simplicity, we can assume that the processor is stalled. Although it may issue further requests, it should not issue a request for the same cache block until the first one is completed. Requests for independent blocks can be handled separately.

■ The write miss is sent to the directory controller for this memory block. Note that although one cache controller handles all the requests for a given cache block, regardless of its memory contents, the directory controller handles requests for different blocks as independent events (assuming sufficient buffering, which is allocated before the directory issues any further messages on behalf of the request). The only conflict at the directory controller is when two requests arrive for the same block. The controller must wait for the first operation to be completed. It can simply NAK the second request or buffer it, but it should not service the second request for a given memory block until the first is completed.

■ Now consider what happens at the directory controller: Suppose the write miss is the next thing to arrive at the directory controller. The controller sends out the invalidates, which can always be accepted after a limited delay by the cache controller. Note that one possibility is that the cache controller has an outstanding miss for the same block. This is the dual case to the snooping scheme, and we must once again break the tie by forcing the cache controller to accept and act on the directory request. Depending on the exact timing, this cache controller will either get the cache line later from the directory or will receive a NAK and have to restart the process.

The case where the block is exclusive is somewhat trickier. Our analysis begins when the write miss arrives at the directory controller for processing. There are two cases to consider:

■ The directory controller sends a fetch/invalidate message to the processor where it arrives to find the block in the exclusive state. The cache controller sends a data write-back to the home directory and makes its state invalid. This reply arrives at the home directory controller, which can always accept the reply, since it preallocated the buffer. The directory controller sends back the data to the requesting processor, which can always accept the reply; after the cache is updated, the requesting cache controller notifies the processor.

■ The directory controller sends a fetch/invalidate message to the node indicated as owner. When the message arrives at the owner node, it finds that this cache controller has taken a read or write miss that caused the block to be replaced. In this case, the cache controller has already sent the block to the home directory with a data write-back and made the data unavailable. Since this is exactly the effect of the fetch/invalidate message, the protocol operates correctly in this case as well.

We have shown that our coherence mechanism operates correctly when the cache and directory controller can accept requests for operation on cache blocks for which they have no outstanding operations in progress, when replies are always accepted, and when requests can be NAKed and forced to retry. Like the case of the snooping protocol, the cache controller must be able to break ties, and it always does so by favoring the instructions from the directory. The ability to NAK requests is what allows an implementation with finite buffering to avoid deadlock.

Implementing the Directory Controller

To implement a cache coherence scheme, the cache controller must have the same abilities it needed in the snooping case, namely, the capability of handling requests for independent blocks while awaiting a response to a request from the local processor. The incoming requests are still processed in order, and each one is completed before beginning the next. Should a cache controller receive too many requests in a short period of time, it can NAK them, knowing that the directory will subsequently regenerate the request.

The directory must also be multithreaded and able to handle requests for multiple blocks independently. This situation is somewhat different than having the cache controller handle incoming requests for independent blocks, since the directory controller will need to begin processing one request while an earlier one is still underway. The directory controller cannot wait for one to complete before servicing the next request, since this could lead to deadlock. Instead, the directory controller must be *reentrant*; that is, it must be capable of suspending its execution

while waiting for a reply and accepting another transaction. The only place this must occur is in response to read or write misses, while waiting for a response from the owner. This leads to three important observations:

1. The state of the controller need only be saved and restored while either a fetch operation from a remote location or a fetch/invalidate is outstanding.

2. The implementation can bound the number of outstanding transactions being handled in the directory by simply NAKing read or write miss requests that could cause the number of outstanding requests to be exceeded.

3. If instead of returning the data through the directory, the owner node forwards the data directly to the requester (as well as returning it to the directory), we can eliminate the need for the directory to handle more than one outstanding request. This motivation, in addition to the reduction of latency, is the reason for using the forwarding style of protocol. There are other complexities from forwarding protocols that arise when requests arrive closely spaced in time.

The major remaining implementation difficulty is to handle NAKs. One alternative is for each processor to keep track of its outstanding transactions so it knows, when the NAK is received, what the requested transaction was. The alternative is to bundle the original request into the NAK, so that the controller receiving the NAK can determine what the original request was. Because every request allocates a slot to receive a reply and a NAK is a reply, NAKs can always be received. In fact, the buffer holding the return slot for the request can also hold information about the request, allowing the processor to reissue the request if it is NAKed.

In practice, great care is required to implement these protocols correctly and to avoid deadlock. The key ideas we have seen in this section—dealing with nonatomicity and finite buffering—are critical to ensuring a correct implementation. Designers have found that both formal and informal verification techniques are helpful for ensuring that implementations are correct.

I.8   The Custom Cluster Approach: Blue Gene/L

Blue Gene/L (BG/L) is a scalable message-passing supercomputer whose design offers unprecedented computing density as measured by compute power per watt. By focusing on power efficiency, BG/L also achieves unmatched throughput per cubic foot. High computing density, combined with cost-effective nodes and extensive support for RAS, allows BG/L to efficiently scale to very large processor counts.

BG/L is a distributed-memory, message-passing computer but one that is quite different from the cluster-based, often throughput-oriented computers that rely on commodity technology in the processors, interconnect, and, sometimes, the packaging and system-level organization. BG/L uses a special customized processing node that contains two processors (derived from low-power, lower-clock-rate PowerPC 440 chips used in the embedded market), caches, and interconnect logic.

A complete computing node is formed by adding SDRAM chips, which are the only commodity semiconductor parts in the BG/L design.

BG/L consists of up to 64 K nodes organized into 32 racks each containing 1 K nodes in about 50 cubic feet. Each rack contains two double-sided boards with 512 nodes each. Due to the high density within a board and rack, 85% of the interconnect is within a single rack, greatly reducing the complexity and latency associated with connections between racks. Furthermore, the compact size of a rack, which is enabled by the low power and high density of each node, greatly improves efficiency, since the interconnection network for connections within a single rack are integrated into the single compute chip that comprises each node.

Appendix F discusses the main BL/G interconnect network, which is a three-dimensional torus. There are four other networks: Gigabit Ethernet, connected at designated I/O nodes; a JTAG network used for test; a barrier network; and a global collective network. The barrier network contains four independent channels and can be used for performing a global or or a global and across all the processors with latency of less than 1.5 microseconds. The global collective network connects all the processors in a tree and is used for global operations. It supports a variety of integer reductions directly, avoiding the need to involve the processor, and leading to times for large-scale reductions that are 10 to 100 times faster than in typical supercomputers. The collective network can also be used to broadcast a single value efficiently. Support for the collective network as well as the torus is included in the chip that forms of the heart of each processing node.

The Blue Gene/L Computing Node

Each BG/L node consists of a single processing chip and several SDRAM chips. The BG/L processing chip, shown in Figure I.18, contains the following:

1. Two PowerPC 440 CPUs, each a two-issue superscalar with a seven-stage pipeline and speculative out-order issue capability, clocked at a modest (and power-saving) 700 MHz. Each CPU has separate 32 KB I and D caches that are nonbblocking with up to four outstanding misses. Cache coherence must be enforced in software. Each CPU also contains a pair of floating-point coprocessors, each with its own FP register set and each capable of issuing a multiply-add each clock cycle, supporting a special SIMD instruction set capability that includes complex arithmetic using a pair of registers and 128-bit operands.

2. Separate fully associative L2 caches, each with 2 KB of data and a 128-byte block size, that act essentially like prefetch buffers. The L2 cache controllers recognize streamed data access and also handle prefetch from L3 or main memory. They have low latency (11 cycles) and provide high bandwidth (5 bytes per clock). The L2 prefetch buffer can supply 5.5 GB/sec to the L1 caches.

3. A 4 MB L3 cache implemented with embedded DRAM. Each L2 buffer is connected by a bus supplying 11 GB/sec of bandwidth from the L3 cache.
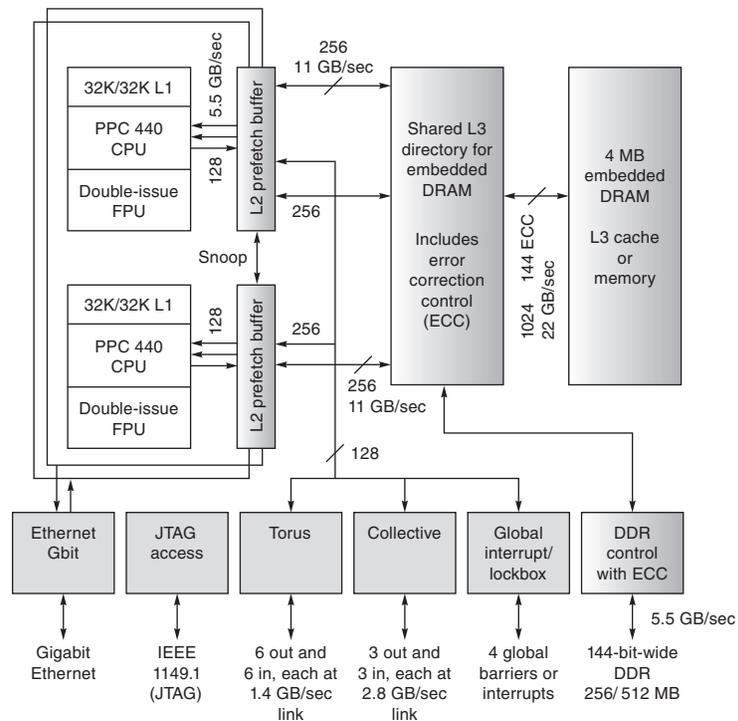
Figure I.18 The BG/L processing node. The unfilled boxes are the PowerPC processors with added floating-point units. The solid gray boxes are network interfaces, and the shaded lighter gray boxes are part of the memory system, which is supplemented by DDR RAMS.

4. A memory bus supporting 256 to 512 MB of DDR DRAMS and providing 5.5 GB/sec of memory bandwidth to the L3 cache. This amount of memory might seem rather modest for each node, given that the node contains two processors, each with two FP units. Indeed Amdahl's rule of thumb (1 MB per 1 MIPS) and an assumption of 25% of peak performance would favor about 2.7 times the memory per node. For floating-point-intensive applications where the computational need usually grows faster than linear in the memory size, the upper limit of 512 MB/node is probably reasonable.

5. Support logic for the five interconnection networks.

By placing all the logic other than DRAMs into a single chip, BG/L achieves higher density, lower power, and lower cost, making it possible to pack the processing nodes extremely densely. The density in terms allows the interconnection networks to be low latency, high bandwidth, and quite cost effective. The combination yields a supercomputer that scales very cost-effectively, yielding an order-of-magnitude improvement

Figure I.19 **The 64 K-processor Blue Gene/L system.**

in GFLOPs/watt over other approaches as well as significant improvements in GFLOPS/$ for very large-scale multiprocessors.

For example, BG/L with 64 K nodes has a peak performance of 360 TF and uses about 1.4 megawatts. To achieve 360 TF peak using the Power5+, which is the most power-efficient, high-end FP processor, would require about 23,500 processors (the dual processor can execute up to 8 FLOPs/clock at 1.9 GHz). The power requirement for just the processors, without external cache, DRAM, or interconnect, would be about 2.9 megawatts, or about double the power of the entire BG/L system. Likewise, the smaller die size of the BG/L node and its need for DRAMs as the only external chip produce significant cost savings versus a node built using a high-end multiprocessor. Figure I.19 shows a photo of the 64K node BG/L. The total size occupied by this 128K-processor multiprocessor is comparable to that occupied by earlier multiprocessors with 16K processors.

I.9 Concluding Remarks

The landscape of large-scale multiprocessors has changed dramatically over the past five to ten years. While some form of clustering is now used for all the largest-scale multiprocessors, calling them all "clusters" ignores significant differences in architecture, implementation style, cost, and performance. Bell and Gray

Terminology	Characteristics	Examples
MPP	Originally referred to a class of architectures characterized by large numbers of small, typically custom processors and usually using an SIMD style architecture.	Connection Machines CM-2
SMP (symmetric multiprocessor)	Shared-memory multiprocessors with a symmetric relationship to memory; also called UMA (uniform memory access). Scalable versions of these architectures used multistage interconnection networks, typically configured with at most 64 to 128 processors.	SUN Sunfire, NEC Earth Simulator
DSM (distributed shared memory)	A class of architectures that support scalable shared memory in a distributed fashion. These architectures are available both with and without cache coherence and typically can support hundreds to thousands of processors.	SGI Origin and Altix, Cray T3E, Cray X1, IBM p5 590/5
Cluster	A class of multiprocessors using message passing. The individual nodes are either commodities or customized, likewise the interconnect.	See commodity and custom clusters
Commodity cluster	A class of clusters where the nodes are truly commodities, typically headless workstations, motherboards, or blade servers, connected with a SAN or LAN usually accessible via an I/O bus.	"Beowulf" and other "homemade" clusters
Custom cluster	A cluster architecture where the nodes and the interconnect are customized and more tightly integrated than in a commodity cluster. Also called distributed memory or message passing multiprocessors.	IBM Blue Gene, Cray XT3
Constellation	Large-scale multiprocessors that use clustering of smaller-scale multiprocessors, typically with a DSM or SMP architecture and 32 or more processors.	Larger SGI Origin/Altix, ASC Purple

Figure I.20 A classification of large-scale multiprocessors. The term *MPP,* which had the original meaning described above, has been used more recently, and less precisely, to refer to all large-scale multiprocessors. None of the commercial shipping multiprocessors is a true MPP in the original sense of the word, but such an approach may make sense in the future. Both the SMP and DSM class includes multiprocessors with vector support. The term *constellation* has been used in different ways; the above usage seems both intuitive and precise [Dongarra et al. 2005].

[2002] discussed this trend, arguing that clusters will dominate. While Dongarra et al. [2005] agreed that some form of clustering is almost inevitable in the largest multiprocessors, they developed a more nuanced classification that attempts to distinguish among a variety of different approaches.

In Figure I.20 we summarize the range of terminology that has been used for large-scale multiprocessors and focus on defining the terms from an architectural and implementation perspective. Figure I.21 shows the hierarchical relationship of these different architecture approaches. Although there has been some convergence in architectural approaches over the past 15 years, the TOP500 list, which reports the 500 fastest computers in the world as measured by the Linpack benchmark, includes commodity clusters, customized clusters, Symmetric Multiprocessors (SMPs), DSMs, and constellations, as well as processors that are both scalar and vector.
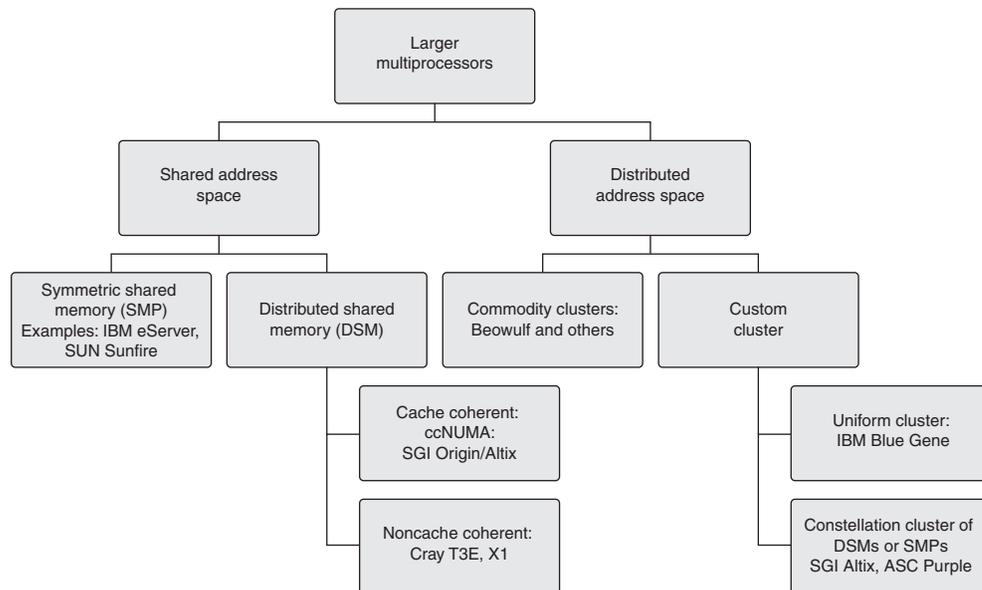
Figure I.21 The space of large-scale multiprocessors and the relation of different classes.

Nonetheless, there are some clearly emerging trends, which we can see by looking at the distribution of types of multiprocessors in the TOP500 list:

1. Clusters represent a majority of the systems. The lower development effort for clusters has clearly been a driving force in making them more popular. The high-end multiprocessor market has not grown sufficiently large to support full-scale, highly customized designs as the dominant choice.

2. The majority of the clusters are commodity clusters, often put together by users, rather than a system vendor designing a standard product.

3. Although commodity clusters dominate in their representation, the top 25 entries on the list are much more varied and include 9 custom clusters (primarily instances of Blue Gene or Cray XT3 systems), 2 constellations, 8 commodity clusters, 2 SMPs (one of which is the NEC Earth Simulator, which has nodes with vector processors), and 4 DSM multiprocessors.

4. Vector processors, which once dominated the list, have almost disappeared.

5. The IBM Blue Gene dominates the top 10 systems, showing the advantage of an approach the uses some commodity processor cores, but customizes many other functions and balances performance, power, and packaging density.

6. Architectural convergence has been driven more by market effects (lack of growth, limited suppliers, etc.) than by a clear-cut consensus on the best architectural approaches.

Software, both applications and programming languages and environments, remains the big challenge for parallel computing, just as it was 30 years ago, when multiprocessors such as the Illiac IV were being designed. The combination of ease of programming with high parallel performance remains elusive. Until better progress is made on this front, convergence toward a single programming model and underlying architectural approach (remembering that for uniprocessors we essentially have one programming model and one architectural approach!) will be slow or will be driven by factors other than proven architectural superiority.