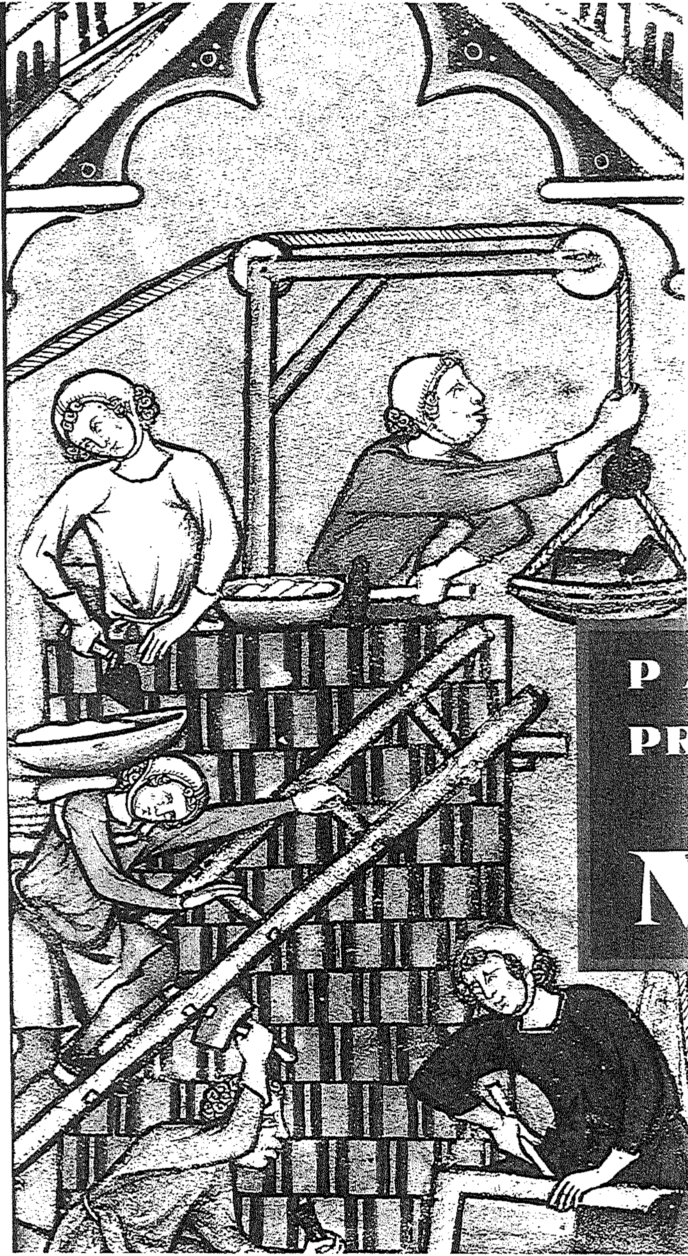
Peter S. Pacheco
PARALLEL
PROGRAMMING
with
MPI

CHAPTER 2

An Overview of Parallel Computing

THIS CHAPTER CONTAINS A BRIEF SURVEY of parallel computing. In it we'll discuss the architecture of current parallel systems and try to give a short overview of the current state of the art in methods for programming these systems. The chapter is mostly independent of the rest of the book. It can be used as a reference chapter if some of the hardware/software issues that arise in later chapters are not completely clear. However, since message passing may not be the last word in parallel computing, we suggest that you look it over so that you'll have an idea of some of the major ideas and issues in parallel computing.

2.1 Hardware

There are as many varieties of parallel computing hardware as there are stars in the sky ... well, not quite, but there are *many* different architectures, and trying to impose some logical order on them may strike some as rather akin to Procrustes' attempts to extend hospitality to his visitors.[1] But we must persevere.

1 "Procrustes or the Stretcher ... had an iron bedstead, on which he used to tie all travellers who fell into his hands. If they were shorter than the bed, he stretched their limbs to make them fit it; if they were longer than the bed, he lopped off a portion. Theseus served him as he had served others" [5].

2.1.1 Flynn's Taxonomy

The original classification of parallel computers is popularly known as **Flynn's taxonomy.** In 1966 Michael Flynn classified systems according to the number of instruction streams and the number of data streams. The classical von Neumann machine has a single instruction stream and a single data stream, and hence is identified as a **single-instruction single-data (SISD)** machine. At the opposite extreme is the **multiple-instruction multiple-data (MIMD)** system, in which a collection of autonomous processors operate on their own data streams. In Flynn's taxonomy, this is the most general architecture. Intermediate between SISD and MIMD systems are SIMD and MISD systems. We'll discuss each of these architectures.

2.1.2 The Classical von Neumann Machine

The classical von Neumann machine is divided into a CPU and main memory. The CPU is further divided into a control unit and an arithmetic-logic unit (ALU). The memory stores both instructions and data. The control unit directs the execution of programs, and the ALU carries out the calculations called for in the program. When they are being used by the program, instructions and data are stored in very fast memory locations, called **registers.** Of course, fast memory is more expensive, so there are relatively few registers.

Both data and program instructions are moved between memory and the registers in the CPU. The route along which they travel is called a **bus.** It's basically a collection of parallel wires together with some hardware that controls access to the bus. Faster busses will have more wires; e.g., a 32-bit bus is faster than a 16-bit bus.

The classical von Neumann machine needs some additional devices before it can be useful: input and output devices, and usually extended storage devices such as a hard disk.

The **von Neumann bottleneck** is the transfer of data and instructions between memory and the CPU: no matter how fast we make our CPUs, the speed of execution of programs is limited by the rate at which we can transfer the (inherently sequential) sequence of instructions and data between memory and the CPU. As a result, few computers today are strictly classical von Neumann machines. For example, most machines now have a hierarchical memory: in addition to the main memory and registers, there is an intermediate memory, faster than main memory but slower than registers, called **cache**. The idea behind cache is the observation that programs tend to access both data and instructions sequentially. Hence, if we store a small block of data and a small block of instructions in fast memory, most of the program's memory accesses will use the fast memory rather than the slower main memory.

2.1.3 Pipeline and Vector Architectures

The first widely used extension to the basic von Neumann model was **pipelining.** If the various circuits in the CPU are split up into functional units, and the functional units are set up in a pipeline, then the pipeline can, in theory, produce a result during each instruction cycle. As an example, suppose we have a program containing the following code:

```
float x[100], y[100], z[100];
for (i = 0; i < 100; i++)
    z[i] = x[i] + y[i];
```

Further suppose that a single addition consists of the following sequence of operations:

1. Fetch the operands from memory.
2. Compare exponents.
3. Shift one operand.
4. Add.
5. Normalize the result.
6. Store result in memory.

Now, suppose we have functional units that perform each of these basic operations, and these functional units are arranged in a pipeline. That is, the output of one functional unit is the input to the next. Then, while, say, x[0] and y[0] are being added, one of x[1] and y[1] can be shifted, the exponents in x[2] and y[2] can be compared, and x[3] and y[3] can be fetched. Thus, once the pipeline is "full," we can produce a result six times faster than we could without the pipelining.

A further improvement can be obtained by adding *vector* instructions to the basic machine instruction set. In our example of adding 100 pairs of floats, if we don't have vector instructions, an instruction corresponding to each of our basic operations will have to be fetched and decoded 100 times. With vector instructions, each of the basic instructions only needs to be issued once. The difference is somewhat analogous to the difference between the Fortran 77 code

```
      do 100 i = 1, 100
          z(i) = x(i) + y(i)
  100 continue
```

and the equivalent Fortran 90 code

```
      z(1:100) = x(1:100) + y(1:100)
```

Another improvement in vector machines is the use of multiple memory banks: operations that access main memory (fetch or store) are several times slower than operations that only involve the CPU (e.g., add). The use of independent memory banks can, to a degree, overcome this problem. For example, suppose that we can execute a CPU operation once every CPU cycle, but we can only execute a memory access every four cycles. Then if we have four memory banks, and our data is properly distributed among the banks, we can access memory once per cycle. In our example, if, say, `z[i]` is stored in memory bank i mod 4, then we can execute one store operation per cycle.

Some authors regard vector processors as MISD machines; others state that there is no such thing as an MISD machine, and that these machines are a variant of SIMD machines. Still others say that they aren't really parallel machines at all.

Some examples of vector processors are the CRAY C90 and the NEC SX4. See the references at the end of the chapter for information on performance benchmarks.

The great virtue of vector processors is that they are well understood and there are extremely good compilers. So it is relatively easy to write programs that obtain very high performance, and, as a consequence, they continue to be very popular for high-performance scientific computing.

There are, however, several drawbacks. The principles of pipelining and vectorization don't work well for programs that use irregular structures or use many branches—the key to performance is filling the pipeline and keeping it full. If operands aren't laid out properly in memory, this is impossible. Further, if a program has lots of conditional branches, there will be little opportunity for the use of vector instructions. Perhaps the greatest drawback is that they don't seem to *scale* well. That is, it's not clear how to modify them so that they can handle ever larger problems. Even if we add several pipelines and manage to keep them full, the upper limit on their speed will be some small multiple of the speed of the CPU.

2.1.4 SIMD Systems

A pure SIMD system (as opposed to a vector processor) has a single CPU devoted exclusively to control, and a large collection of subordinate ALUs, each with its own (small amount of) memory. During each instruction cycle, the control processor broadcasts an instruction to all of the subordinate processors, and each of the subordinate processors either executes the instruction or is idle. For example, suppose we have three arrays x, y, and z, distributed so that the memory of each processor contains one element of each array. Now suppose that we want to execute the following sequence of (serial) instructions:

```
for (i = 0; i < 1000; i++)
    if (y[i] != 0.0)
        z[i] = x[i]/y[i];
    else
```


Then each subordinate processor would execute something like the following sequence of operations:

Time Step 1. Test `local_y != 0.0`.

Time Step 2.

a. If `local_y` was nonzero, `z[i] = x[i]/y[i]`.
b. If `local_y` was zero, do nothing.

Time Step 3.

a. If `local_y` was nonzero, do nothing.
b. If `local_y` was zero, `z[i] = x[i]`.

Note that this implies completely synchronous execution of statements. In other words, at any given instant of time, a given subordinate process is either "active" and doing exactly the same thing as all the other active processes, or it is idle.

The example makes the disadvantages of an SIMD system clear: in a program with many conditional branches or long segments of code whose execution depends on conditionals, it's entirely possible that many processes will remain idle for long periods of time.

However, the example doesn't make clear that SIMD machines tend to be relatively easy to program if the underlying problem has a regular structure. Furthermore, although communication is quite expensive in distributed-memory MIMD systems, it is basically no more expensive than computation in SIMD machines. (We'll try to explain why this is so, after we've talked about MIMD systems.) Finally, they do scale well, as the following examples show.

The most famous examples of SIMD machines are the CM-1 and CM-2 Connection Machines that were produced by Thinking Machines. The CM-2 had up to 65,536 1-bit processors and up to 8 billion bytes of memory. Maspar also produced SIMD machines. The MP-2 has up to 16,384 32-bit ALUs and up to 4 billion bytes of memory.

2.1.5 General MIMD Systems

The key difference between MIMD and SIMD systems is that with MIMD systems, the processors are autonomous: each processor is a full-fledged CPU with both a control unit and an ALU. Thus each processor is capable of executing its own program at its own pace. In particular, unlike SIMD machines, MIMD systems are *asynchronous*. There is often no global clock, and, unless, the processors are specifically programmed to synchronize with each other, there may be no correspondence between what is being done on different processors—even if the processors are executing the same program.

The world of MIMD systems is divided into shared-memory and distributed-memory systems. Some authors distinguish between the two architectures by

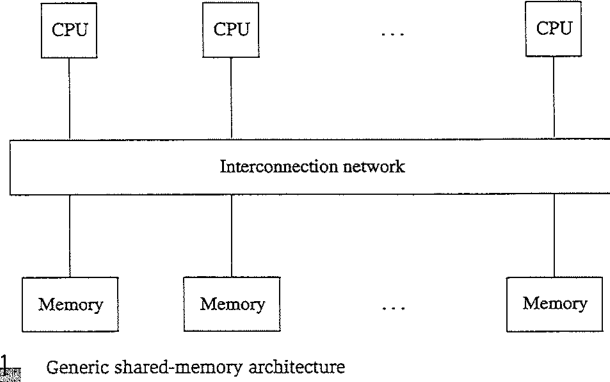

Figure 2.1 Generic shared-memory architecture

calling shared-memory systems **multiprocessors** and distributed-memory systems **multicomputers.** However, this terminology hasn't gained universal acceptance, and it is quite common to hear "multiprocessor" used as a synonym for "parallel processor."

2.1.6 Shared-Memory MIMD

As the name implies, the generic shared-memory machine consists of a collection of processors and memory modules interconnected by a network (see Figure 2.1).

Bus-Based Architectures

The simplest interconnection network is bus based. However, if multiple processors are simultaneously attempting to access memory, the bus will become saturated, and there may be long delays between starting a fetch or store and actually copying the data. Thus each processor usually has access to a fairly large cache (see Figure 2.2). Because of the limited bandwidth of a bus, these architectures do not scale to large numbers of processors. For example, the largest configuration of the currently popular SGI Challenge XL has only 36 processors.

Switch-Based Architectures

Most other shared-memory architectures rely on some type of switch-based interconnection network. As an example, the basic unit of the Convex SPP1200 is a 5×5 **crossbar switch.** A crossbar can be visualized as a rectangular mesh of wires with switches at the points of intersection, and terminals on its left and top edges. Processors or memory modules can be connected to the terminals (see Figure 2.3). The switches can either allow a signal to pass through in both the vertical and horizontal directions simultaneously, or they can redirect a signal from vertical to horizontal or vice versa (see Figure 2.4). Thus, for example, if we have processors on the left and memory modules on the top of the crossbar, then any processor can access any memory module. Further, any other processor can simultaneously access any other memory module. That is, communication between two units will not interfere with communication between any other two units. So crossbar switches don't suffer from the problems of saturation that we encountered with busses.

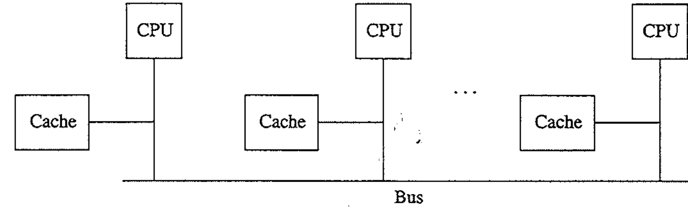

Figure 2.2 Bus-based shared-memory architecture

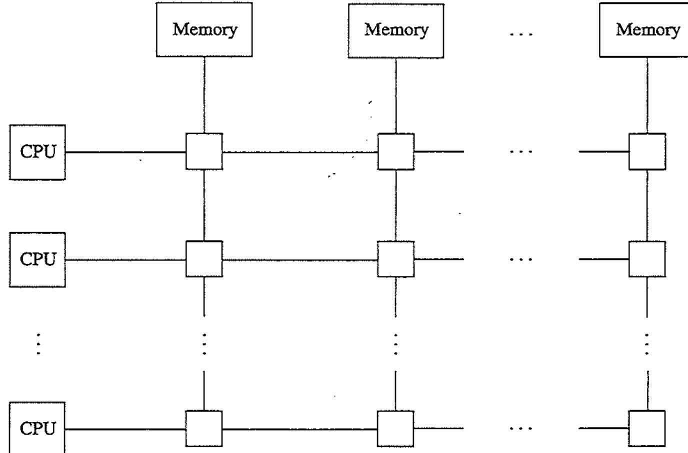

Figure 2.3 Crossbar switch

Unfortunately, they tend to be very expensive: an $m \times n$ crossbar will need mn hardware switches. Thus, they tend to be fairly small. For example,

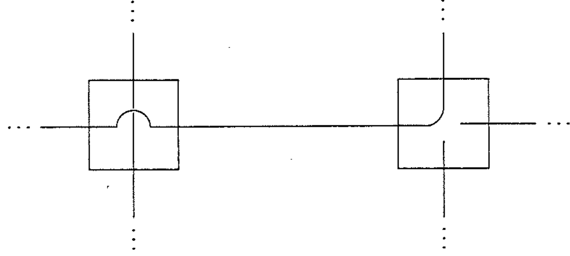

Figure 2.4 Configurations of the internal switches

in the Convex SPP1200, in order to have more than eight processors in a single machine, two or more crossbars are connected in a ring.

Note that this implies that when a processor accesses memory attached to another crossbar, the access times will be greater. This is, of course, undesirable, but it is a compromise that has been reached by virtually all designers of shared-memory machines. That is, nonuniform access times are the rule rather than the exception. Such systems are called **nonuniform memory access** or NUMA systems.

Cache Coherence

A problem that is encountered with any shared-memory architecture that allows the caching of shared variables is *cache consistency* or *cache coherence.* If a processor accesses a shared variable in its cache, how will it know whether the value stored in the variable is current? That is, suppose processor A wants to access a shared variable x in its cache. How does A know that some other process B hasn't modified its copy of x, rendering A's copy out of date? There are a number of cache consistency protocols, and they vary considerably in complexity. The simplest is probably the **snoopy protocol,** and it is suitable for small bus-based machines. The basic idea is that in addition to the usual hardware associated with a CPU, each processor has a cache controller. Among other things, the cache controllers "snoop" on the bus; i.e., they monitor the bus traffic. When a processor updates a shared variable, it also updates the corresponding main memory location. The cache controllers on the other processors detect the write to main memory and mark their copies of the variable as invalid. Notice that the bus makes this possible: any traffic on the bus can be monitored by all the controllers. Thus this approach is unsuitable for other types of shared-memory machines.

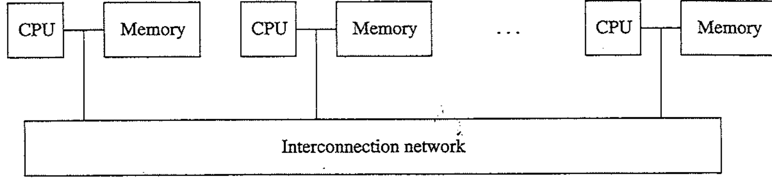

Figure 2.5 Generic distributed-memory system

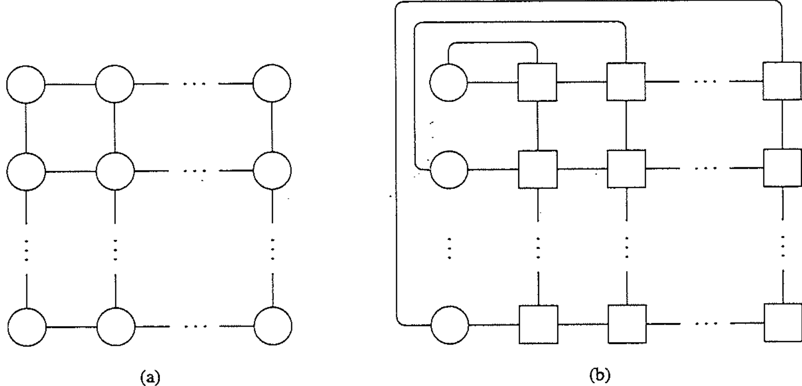

Figure 2.6 Different types of distributed-memory systems: (a) a static network (mesh) and (b) a dynamic network (crossbar)

2.1.7 Distributed-Memory MIMD

In distributed-memory systems, each processor has its own private memory. Thus, a generic distributed-memory system can be represented as in Figure 2.5. If we view a distributed-memory system as a graph, where the edges are communication wires, then there are two broad types of graphs: those in which each vertex corresponds to a processor/memory pair, or **node,** and those in which some vertices correspond to nodes and others correspond to switches. Figure 2.6 illustrates the distinction: round vertices are nodes and square vertices are switches. Networks of the first type are called **static networks** and networks of the second are called **dynamic networks.**

From a performance and programming standpoint, the ideal interconnection network is a fully connected network, in which each node is directly connected to every other node (see Figure 2.7). With a fully connected network, each node can communicate directly with every other node. Furthermore, the

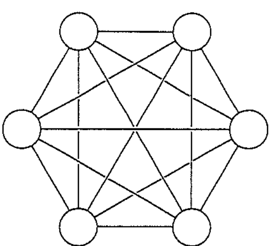

Figure 2.7 A fully connected interconnection network

communication involves no delay, and any node can communicate with any other node at the same time that any other communication is taking place. Unfortunately, the cost of such a network makes it impractical to construct such a machine with more than a few nodes.

Dynamic Interconnection Networks

Perhaps the closest we can come, in practice, to a fully connected network is a crossbar switch in which each process is connected to a terminal on the left edge and a terminal on the right edge (see the illustration on the right in Figure 2.6). Essentially the only delay in communication comes from the setting of a single switch, and if node i is communicating with node j, then any other pair of distinct nodes can communicate simultaneously. However, as we noted in section 2.1.6, these networks are also very expensive, and it is unusual to see crossbars with more than 16 processors. A notable exception is the Fujitsu VPP 500, which uses a 224 × 224 crossbar with 224 nodes.

A less expensive solution is to use a **multistage** switching network. There are a number of different types of multistage network. An example, an **omega** network, is illustrated in Figure 2.8. If we have p nodes, then an omega network will use $p\log_2(p)/2$ switches, and, as a consequence, is a good deal less expensive than the crossbar, which uses p^2 switches. With the omega network, any node can communicate with any other node. However, there is a relatively high probability that communication between two nodes will interfere with communication between two other nodes. Further, the delay in transmitting a message is increased, since $\log_2(p)$ switches must be set. In its SP series of computers, IBM has compromised between the two switching strategies: it uses an omega network, but the individual switches (represented in Figure 2.8 as 2 × 2 crossbars) are 8 × 8 crossbars. Currently the largest installed machine has 512 nodes.

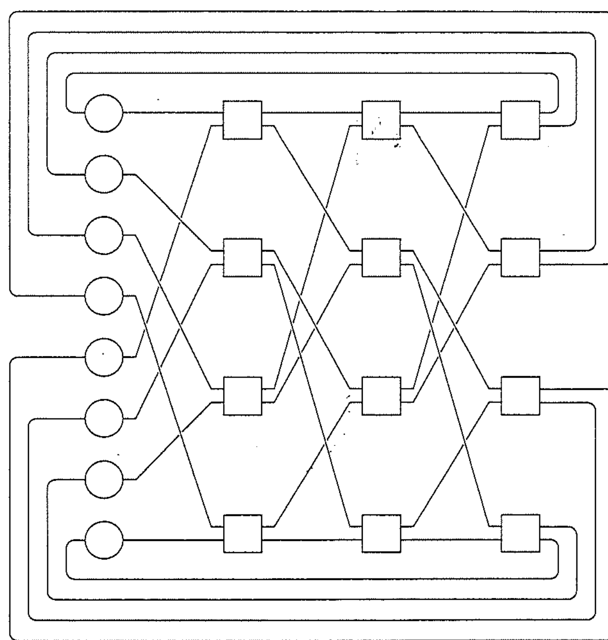

Figure 2.8 An omega network

Static Interconnection Networks

At the opposite extreme from a fully connected network is a **linear array**, a static network in which all but two of the nodes have two immediately adjacent neighboring nodes. A **ring** is a slightly more powerful network. This is just a linear array in which the "terminal" nodes have been joined (see Figure 2.9). The virtue of these networks is that they are relatively inexpensive: beyond the cost of the nodes, there is only an additional cost of $p-1$ or p wires. They also scale well: it's quite easy and inexpensive to increase the size of the network so that it includes arbitrarily many nodes. The principal drawback is that the number of available wires is extremely limited: if two nodes are communicating, it's very likely that other nodes attempting to communicate will be unable to do so. Furthermore, in a linear array, two processes that are attempting to communicate may have to forward the message along as many as $p-1$ wires, and in a ring it may be necessary to forward the message along as many as $p/2$ wires.

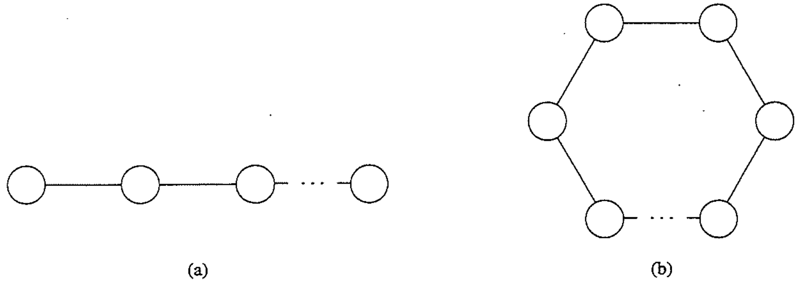

Figure 2.9 (a) A linear array and (b) a ring

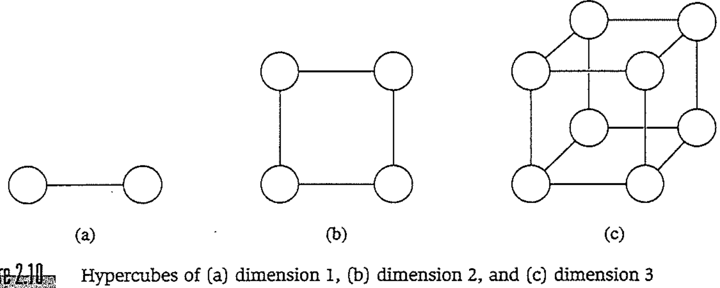

Figure 2.10 Hypercubes of (a) dimension 1, (b) dimension 2, and (c) dimension 3

The practical static interconnection network that is closest to the fully connected network is the **hypercube**. Hypercubes are defined inductively. A dimension 0 hypercube consists of a single node. In order to construct a hypercube of dimension $d > 0$, we take two hypercubes of dimension $d - 1$ and join the corresponding nodes with communication wires. Hypercubes of dimensions 1, 2, and 3 are illustrated in Figure 2.10. Since we double the number of nodes with each increase in dimension, a hypercube of dimension d will contain $p = 2^d$ nodes. Since we add a wire to each node when we increase the dimension by one, in a hypercube of dimension d, each node is directly connected to d other nodes. Thus, it is relatively easy (compared to the linear array or the omega network) to arrange that communicating nodes don't interfere with other communications. Furthermore, it's not difficult to show that if we follow a shortest path between any two nodes in a hypercube of dimension d, then we'll traverse at most d wires (use induction on the dimension). Thus, the maximum number of wires a message will need to be forwarded along is $d = \log_2(p)$ wires. This is much better than the linear array or ring. The principal drawback to the hypercube is its relative lack

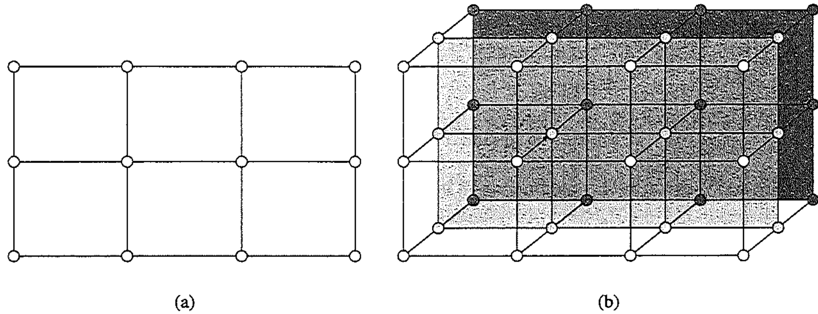

Figure 2.11 (a) Two-dimensional mesh, (b) three-dimensional mesh, and (c) two-dimensional torus

of scalability. In spite of the fact that the first "massively parallel" MIMD system was a hypercube (an nCUBE 10 with 1024 nodes), each time we wish to increase the machine size, we must *double* the number of nodes and add a new wire to each node.

Intermediate between hypercubes and linear arrays are meshes and tori, which are simply higher dimensional analogs of linear arrays and rings, respectively. Figure 2.11 illustrates a two-dimensional mesh, a two-dimensional torus, and a three-dimensional mesh. Observe that an n-dimensional torus can be obtained from an n-dimensional mesh by adding "wrap-around" wires to the nodes on the border. Also observe that, as we increase the dimension, it becomes less and less likely that two pairs of communicating nodes will interfere with each other, and that if a mesh has dimensions $d_1 \times d_2 \times \cdots \times d_n$, then

the maximum number of wires a message will have to traverse is

$$\sum_{i=1}^{n}(d_i - 1).$$

So if a mesh is square, i.e., $d_1 = d_2 = \cdots = d_n$, the maximum will be $n(p^{1/n} - 1)$. More or less the same reasoning applies to tori; e.g., in a "square" torus, the maximum will be $\frac{1}{2}np^{1/n}$. Furthermore, meshes and tori scale better than hypercubes (although not as well as linear arrays and rings). For example, if we wish to increase the size of a $q \times q$ mesh, we simply add a $q \times 1$ mesh and q wires. More generally, we need to add $p^{(n-1)/n}$ nodes if we wish to increase the size of a square n-dimensional mesh or torus. Meshes and tori are currently quite popular. The Intel Paragon is a two-dimensional mesh, and the Cray T3E is a three-dimensional torus. Both scale to thousands of nodes.

Bus-Based Networks

The last, and probably the simplest, network is a bus. A cluster of workstations on an ethernet provides a popular example. Of course, busses tend to be fairly slow, and even worse, busses, especially ethernets, soon become saturated if there are more than a few nodes or more than absolutely minimal communication. Thus, although they are very useful for program development, currently available bus-based systems don't show much promise for very large-scale applications.

2.1.8 Communication and Routing

An issue that soon appears when we study communication in distributed-memory MIMD systems and larger shared-memory systems is that of routing. If two nodes are not directly connected or if a processor is not directly connected to a memory module, how is data transmitted between the two? Let's take a look at this problem on distributed-memory systems using a static interconnection network. Before proceeding, however, we should note that this problem is not necessarily completely solved with hardware: many systems implement parts of their routing using software.

The problem of routing subsumes two additional subproblems: If there are multiple routes joining the two nodes or processor and memory, how is a route decided on? Is the route chosen always a "shortest" path? Most systems use a deterministic shortest-path routing algorithm. That is, if node A communicates with node B, then the route that the communication uses will always be the same, and there will be no other route that uses fewer wires. This issue arises whether the intermediaries are other nodes or switches.

Another problem in this connection is the question of how intermediate nodes forward communications. There are two basic approaches. In order to understand them, let's suppose that node A is sending a message to node C, and node B lies between A and C. Node B has essentially two choices: it can

Time	*Data* *Node A*	*Node B*	*Node C*
0	z y x w		
1	z y x	w	
2	z y	x w	
3	z	y x w	
4		z y x w	
5		z y x	w
6		z y	x w
7		z	y x w
8			z y x w

Figure 2.12 Store-and-forward routing

Time	*Data* *Node A*	*Node B*	*Node C*
0	z y x w		
1	z y x	w	
2	z y	x	w
3	z	y	x w
4		z	y x w
5			z y x w

Figure 2.13 Cut-through routing

read in the entire message, and then send it to node C, or it can immediately forward each identifiable piece, or **packet**, of the message. The first approach is called **store-and-forward routing.** The second is called **cut-through routing.** Store-and-forward routing is illustrated in Figure 2.12. Cut-through routing is illustrated in Figure 2.13. In the figures, the message is composed of four packets, w, x, y, and z. As we can see, using store-and-forward routing the message takes twice as long as the time it takes to send a message between adjacent nodes, while the time it takes using cut-through routing only adds the time it takes to send a single packet. Furthermore, store-and-forward routing uses considerably more memory on the intermediate nodes, since the entire message must be buffered. Thus, most systems use some variant of cut-through routing.

2.2 Software Issues

The idea of a **process** is a fundamental building block in most paradigms of parallel computing. Intuitively, a process is an instance of a program or a subprogram that is executing more or less autonomously on a physical processor.

A program is parallel if, at any time during its execution, it can comprise more than one process. In order to create useful parallel programs, there must be ways that processes can be specified, created, and destroyed, and there must be ways to coordinate interprocess interaction. In this section, we'll take a brief look at how these issues are addressed in different programming paradigms.

2.2.1 Shared-Memory Programming

Although we conventionally think of a shared-memory system as one in which the processors have more or less equal access to all the memory locations, it is perfectly reasonable to *emulate* shared memory with physically distributed memory if we have a mechanism for creating a global address space. Thus, it may be possible to program a distributed-memory system using shared-memory programming primitives, and this discussion may be applicable to a variety of underlying hardware configurations.

Shared-memory systems typically provide both **static** and **dynamic** process creation. That is, processes can be created at the beginning of program execution by a directive to the operating system, or they can be created during the execution of the program. The best-known dynamic process creation function is `fork`. A typical implementation will allow a process to start another, or **child**, process by calling `fork`. The starting, or **parent**, process can wait for the termination of the child process by calling `join`.

Coordination among processes in shared-memory programs is typically managed by three primitives. The first specifies variables that can be accessed by all the processes. The second prevents processes from improperly accessing shared resources. The third provides a means for synchronizing the processes. To illustrate these ideas, let's look at a very simple example. Suppose that each process has computed a *private* int `private_x`. By a private variable, we mean a variable whose contents are accessible to only one process. Thus, each process has defined a distinct variable `private_x` that cannot be accessed by any of the other processes. The program should compute the sum of these private ints, and a single process will print the sum.

We can use the first primitive to allocate a shared variable `sum`. One approach might be to simply prefix the definition of the variable with the keyword `shared`. So part of the variable definition component of our program might be

```
int private_x;
shared int sum = 0;
```

Things get a little more complicated when we start trying to compute the sum. We can't simply have each process compute

```
sum = sum + private_x;
```

In order to understand why this is a problem, recall that a typical system will

Time	Process 0		Process 1	
0	Fetch sum	= 0	Finish calculation of private_x	
1	Fetch private_x	= 2	Fetch sum	= 0
2	Add	2 + 0	Fetch private_x	= 3
3	Store sum	= 2	Add	3 + 0
4			Store sum	= 3

Figure 2.14 One scenario for shared-memory addition

execute something like the following sequence of machine instructions when it performs the add:

```
Fetch sum into register A
Fetch private_x into register B
Add contents of register B to register A
Store contents of register A in sum
```

Now suppose we have two processes, 0 and 1, the value of process 0's `private_x` is 2, and the value of process 1's `private_x` is 3. Then, depending on when the processes try to execute the addition of `private_x` to `sum`, we can compute a value of 2, 3, or 5 for `sum`. For example, consider the sequences of events depicted in Figure 2.14. Of course, `sum` should be 5, but since the sequences of machine commands making up `sum = sum + private_x` overlapped, the value computed by process 0 was overwritten by process 1.

Thus, we must make sure that the command

```
sum = sum + private_x;
```

is executed by only one process at a time. When we wish to ensure that only one process can execute a certain sequence of statements at a time, we are trying to arrange for **mutual exclusion,** and the sequence of statements is called a **critical section.**

One of the simplest approaches to solving the problem of mutual exclusion is called a **binary semaphore.** The basic idea is that there is a shared variable `s` whose value indicates whether the critical section is free. If `s` is 1, the section is free. If it's 0, the region cannot be accessed. Thus we would like to do something like this on each process:

```
shared int s = 1;

while (!s);              /* Wait until s = 1  */
s = 0;                   /* Close down access */
sum = sum + private_x;   /* Critical section  */
s = 1;                   /* Re-open access    */
```

The problem is that the operations involved in manipulating `s` are not **atomic.** That is, while one process is fetching `s = 1` into a register to test whether it's

OK to enter the critical region, another process can be storing `s = 0`. We need to be able to arrange that once a process starts to access `s`, no other process can access it until the original process is done with the access, including the reset of its value.

Thus, in addition to the shared variable, a binary semaphore consists of two special functions:

```
void P(int*  s  /* in/out */);
void V(int*  s  /* out    */);
```

The first function, `P`, has an effect similar to

```
while (!s);
s = 0;
```

However, it prevents other processes from accessing `s` once one process gets out of the loop. Similarly `V` sets `s` to 1, but it does this "atomically." The mechanics of achieving atomicity are system dependent. A simple solution is the addition of machine commands that "lock" and "unlock" variables: when a variable is locked, only the process that locked it can write to it.

The final issue we need to address is how to make sure that the correct sum is printed. In other words, if, say, process 0 is printing the sum, how can it know when all the processes have completed adding in their `private_xs` to `sum`? In view of the preceding discussion, we could create another shared variable that we could use to maintain a count of the number of processes that have updated `sum`. However, this is usually carried out with a somewhat higher-level operation called a **barrier.** A barrier is usually implemented as a function (which may or may not take an argument). Once a process has called the function, it will not return until every other process has called it. Thus, if we have a barrier after our sum, process 0 will know that the additions have been completed once it returns from the call to the barrier. In summary, then, our program body should look something like this:

```
int private_x;
shared int sum = 0;
shared int s = 1;

/* Compute private_x */
      :
P(&s);
sum = sum + private_x;
V(&s);

Barrier();

if (I'm process 0)
    printf("sum = %d\n", sum);
```


The concept of shared variables and barriers is quite natural and appealing to programmers that have some experience with conventional systems. The idea of a binary semaphore, however, is not so appealing. It is somewhat error-prone and forces serial execution of the critical region. Thus, a number of alternatives have been devised. Monitors provide a higher-level alternative available on many systems. Basically, they encapsulate shared data structures and the operations that can be performed on them. In other words, the shared data structures are defined in the monitor, and the critical regions are functions of the monitor. When a process calls a monitor function, the other processes are prevented from calling the function.

Unfortunately, monitors do nothing to solve the serialization problem. It's not difficult to imagine alternative approaches to our addition program that don't enforce serial access to the shared variables. However, the obvious solutions tend to be extremely complicated in that they introduce other shared variables and critical regions. There are other solutions, but discussion of them is beyond the scope of this brief overview. See the references at the end of the chapter for information on other solutions.

2.2.2 Message Passing

The most commonly used method of programming distributed-memory MIMD systems is message passing, or some variant of message passing. In basic message passing, the processes coordinate their activities by explicitly sending and receiving messages. For example, at its most basic, the Message-Passing Interface (MPI) provides a function for sending a message:

```
int MPI_Send(void*        buffer      /* in */,
             int          count       /* in */,
             MPI_Datatype datatype    /* in */,
             int          destination /* in */,
             int          tag         /* in */,
             MPI_Comm     communicator /* in */)
```

and a function for receiving a message:

```
int MPI_Recv(void*        buffer      /* out */,
             int          count       /* in */,
             MPI_Datatype datatype    /* in */,
             int          source      /* in */,
             int          tag         /* in */,
             MPI_Comm     communicator /* in */,
             MPI_Status*  status      /* out */)
```

The current version of MPI assumes that processes are statically allocated; i.e., the number of processes is set at the beginning of program execution, and no additional processes are created during execution. Each process is assigned a

unique integer rank in the range 0, 1, . . . , $p - 1$, where p is the number of processes.

To illustrate the use of the functions, suppose that process 0 wishes to send the float x to process 1. Then it can call MPI_Send as follows:

```
MPI_Send(&x, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
```

Process 1 needs to call MPI_Recv. In order that the data be received properly, it needs to match the tag and communicator arguments, and the memory available for receiving the message, which is specified by the buffer, count, and datatype parameters, must be at least as large as the message sent. The status parameter returns information on such things as the actual size of the message received. Thus, process 1 can call MPI_Recv as follows:

```
MPI_Recv(&x, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
```

There are several issues that need to be addressed here. First, note that the commands executed by process 0 (MPI_Send) will be different from those executed by process 1 (MPI_Recv). However, this does not mean that the programs need to be different. We can simply include the following conditional branch in our program:

```
if (my_process_rank == 0)
    MPI_Send(&x, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
else if (my_process_rank == 1)
    MPI_Recv(&x, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
```

This approach to programming MIMD systems is called **single-program, multiple-data** (SPMD). In SPMD programs, the effect of running different programs is obtained by the use of conditional branches within the source code. This is the most common approach to programming MIMD systems.

Another issue we need to address is the semantics of the send/receive pairing. Suppose process 0 calls MPI_Send, but process 1 doesn't call MPI_Recv until some time later. Does process 0 simply stop and wait until process 1 calls MPI_Recv? Even worse, suppose process 0 calls MPI_Send, but process 1 fails to call MPI_Recv. Does the program crash or hang? The answers will, in general, depend on the system. The key issue is whether the system software provides for buffering of messages.

Buffering

Let's assume that process 0 and process 1 are running on distinct nodes, say, 0 is running on node A and 1 is running on node B. In this case there are several ways of dealing with the first situation: 0 can send a "request to send" to 1 and wait until it receives a "ready to receive" from 1, at which point it begins transmission of the actual message. Alternatively, the system software can buffer the message. That is, the contents of the message can be copied into

a system-controlled block of memory (on A or B, or both), and 0 can continue executing. When 1 arrives at the point where it is ready to receive the message, the system software simply copies the buffered message into the appropriate memory location controlled by 1. The first approach, i.e., process 0 waits until process 1 is ready, is sometimes called **synchronous** communication. The second approach is called **buffered** communication.

The clear advantage of buffered communication is that the sending process can continue to do useful work if the receiving process isn't ready. Disadvantages are that it uses up system resources that otherwise wouldn't be needed (e.g., the memory for buffering), and, if the receiving process is ready, the communication will actually take longer, since it will involve copying between the buffer and the user program memory locations.

Most systems provide some buffering, but the details vary widely. Some systems attempt to buffer all messages. Others buffer only relatively small messages and use the synchronous protocol for large messages. Others let the user decide whether to buffer messages, and how much space should be set aside for buffering. Some systems buffer messages on the sending node, while others buffer them on the receiving node.

Note that if the system provides buffering of messages, then our second problem—process 0 executes a send, but process 1 doesn't execute a receive—shouldn't cause the program to crash; the contents of the message will simply sit in the system-provided buffer until the program ends. If, on the other hand, the system doesn't provide buffering, process 0 will probably hang; it will wait forever for a "ready to receive" from process 1.

Before proceeding, it should be noted that SIMD systems don't incur the overhead of buffering or waiting since every operation is synchronous across all the processes. Thus, process 0 "knows" process 1 is ready to receive, and the message can be immediately transmitted. It should also be noted that there are other approaches to the problem of how to deal with messages if there is no guarantee of synchronization among the processes. We'll discuss one such approach in section 2.2.4.

Blocking and Nonblocking Communication

We also need to look at what happens if we reverse the arrival at the communication points. That is, suppose process 1 executes the receive, but process 0 doesn't execute the send until some later time. The function we used for the receive, MPI_Recv, is **blocking**. This means that when process 1 calls MPI_Recv, if the message is not available, process 1 will remain idle until it becomes available. Note that this isn't quite the same thing as synchronous communication. In synchronous communication, the two processes directly communicate: process 0 won't begin sending the message until it has received explicit permission from process 1. In blocking communication, it may not be necessary for 0 to receive permission to go ahead with the send. For example,

0 may have already buffered the message when 1 is ready to receive, but the communication line joining the processes might be busy.

Most systems provide an alternative, **nonblocking** receive operation. In MPI, it's called MPI_Irecv. The *I* stands for *immediate*. That is, the process returns "immediately" from the call. It has one more parameter than MPI_Recv: a **request**. If, instead of calling MPI_Recv, process 1 called MPI_Irecv, the call would notify the system that process 1 intended to receive a message from 0 with the properties indicated by the argument. The system would initialize the request argument, and process 1 would return. Then process 1 could perform some other useful work (that didn't depend on the message from process 0) and check back later to see if the message had arrived. It would inform the system which message it was looking for through the request argument that was initialized by the original call to MPI_Irecv.

The use of nonblocking communication can be used to provide dramatic improvements in the performance of message-passing programs. If a node of a parallel system has the ability to simultaneously compute and communicate, the overhead due to communication can be substantially reduced. For example, if each node of a system has a communication coprocessor, then we can start a nonblocking communication (e.g., MPI_Irecv), perform computations that don't depend on the result of the communication, and when the computations are completed, finish the nonblocking operation. While the computations are being carried out, the communications co-processor can do most of the work required by the nonblocking operation. Since communication is very expensive relative to computation, overlapping communication and computation can result in tremendous performance gains.

2.2.3 Data-Parallel Languages

One of the simplest approaches to programming parallel systems is called **data parallelism.** In it, a data structure is distributed among the processes, and the individual processes execute the same instructions on their parts of the data structure. Clearly this approach is extremely well suited to SIMD machines. However, it is also quite common to use it on MIMD systems. One of its most attractive aspects is that for very regular structures it is possible for the user program to simply indicate that the structure should be distributed across the processes, and the compiler will automatically replace the user directive with code that distributes the data and performs the data-parallel operations. Let's look at a very simple example.

As we noted in Chapter 1, HPF is a set of extensions to Fortran 90 designed to make it relatively easy for a programmer to write highly efficient data-parallel programs. Here's a simple example that performs a distributed array addition:

```
      program add_arrays
!HPF$ PROCESSORS p(10)
```



```
      real x(1000), y(1000), z(1000)
!HPF$ ALIGN y(:) WITH x(:)
!HPF$ ALIGN z(:) WITH x(:)
!HPF$ DISTRIBUTE x(BLOCK) ONTO p

C     Initialize x and y
      :
      z = x + y
      end
```

We begin by specifying a collection of 10 abstract processors with the first HPF directive. After defining our arrays, the first ALIGN directive specifies that y should be mapped to the abstract processors in the same way that x is. That is, for each i, y(i) is assigned to abstract processor *q* if and only if x(i) is. The second ALIGN statement has a similar effect on z. The DISTRIBUTE statement specifies which elements of x will be mapped to which abstract processors, and since y and z have been aligned with x, it will automatically map the corresponding elements of y and z. BLOCK specifies that x will be mapped by *blocks* onto the processors. That is, the first 1000/10 = 100 elements will be mapped to the first processor, the next 100 to the second, etc. Once the arrays are distributed and initialized, we can simply add corresponding entries on the appropriate abstract processors with the Fortran 90 array addition statement

```
z = x + y
```

A few observations are in order here. HPF doesn't provide a mechanism for specifying the mapping of abstract processors to physical processors. The actual mapping is usually done at execution time, and most systems provide (nonportable) means for a program to determine what the mapping is.

Explicitly aligning the arrays in the HPF directives will probably result in a more efficient executable program. In our example, the compiler will "know" that there won't be any communication when the addition is carried out. If the arrays weren't explicitly aligned, they might not be mapped in the same way to the processors, and hence communication might be necessary.

Finally, the problem of mapping data structures to processors is, in general, a very difficult one, unless the structure is static and very regular (e.g., a dense matrix). This can be a serious problem in *any* parallel program. However, it is especially problematic in data-parallel programs, where a mapping is specified at compile time. We'll return to the problem of mapping data structures to processes in section 2.2.5 and section 8.4.1.

It should be noted that "data parallel" is used in a somewhat different way in other contexts. It can be used to describe a methodology for designing a parallel program. In this context, it is usually contrasted with control-parallel programming, in which parallelism is obtained by partitioning the control or instructions of the program rather than the data. In general, most parallel programs use both approaches to obtain parallelism

2.2.4 RPC and Active Messages

Although message-passing and data-parallel languages are the most widely used methods for programming distributed-memory systems, there are a number of other approaches to programming these systems. Two that have been very successful are RPC (Remote Procedure Call) and active messages. They share the assumption that the communication among processes should be more general than the simple transmission of data: they provide constructs for processes to execute subprograms on *remote* processors. The similarities end here however. RPC is essentially synchronous. In order to call a "remote procedure," one process, the client process, calls a *stub* procedure that sends an argument list to another process, the server process. The argument list is used by the server process in a call to the actual procedure. After completing the procedure, the (possibly modified) arguments are returned to the client process. The client is idle while it waits for the results to be returned by the server. This inefficiency reflects the origin of RPC: it was originally developed for use in distributed systems. The model environment is a collection of autonomous multitasking computers, and client and server processes are running on different computers. While the client process is waiting for the return of the arguments, its host system can perform useful work on other jobs. Clearly this will result in inefficiencies if the host systems are dedicated processors.

Active messages remedy this problem by eliminating the synchronous behavior of the process interaction. The message sent by the source process contains, in its header, the address of a handler residing on the receiving process's processor. When the message arrives, the receiving process is notified via an interrupt, and it runs the handler. The arguments of the handler are the contents of the message. Thus, there is no synchronicity: The first process "deposits" its message in the network and proceeds with its computations. Whenever the message ultimately arrives on the receiving process, the receiving process is interrupted, the handler invoked, and the receiving process continues its work. Thus, active messages provide features of both RPC and nonblocking message passing.

2.2.5 Data Mapping

The issue of **data locality** came up several times in our discussion of data-parallel programming. It is also a critical issue in the programming of both distributed-memory systems and nonuniform memory access (NUMA) shared-memory systems. In general, communication is much more expensive than computation. In conventional systems, it is almost a commonplace that instructions that access memory are much slower than operations that only involve the CPU. This difference in cost is even more dramatic if the memory is remote; i.e., if it is the local memory of another node in a distributed-memory system or if it is a "distant" memory module in a NUMA shared-memory system. Thus, considerable effort has been devoted to the problem of optimal data mapping, that is, the problem of how to assign data elements to processors so that communication is minimized. There is an easy (silly) solution: on a distributed-memory system map all the data-elements to the memory of a single node and have that process do *all* the calculations. (A similar mapping applies to NUMA shared-memory systems.) Of course, this would result in a considerable waste of computational resources. So the problem of **load balancing** is counterpoised to our data locality problem. That is, we want to assign the same amount of work to each processor, or else we'll be wasting our computation resources. Any mapping must take into consideration both load balance *and* data locality.

In this section, we'll take a brief look at what is probably the simplest case of the mapping problem: how to map a linear array to a collection of nodes in a distributed-memory system. For the sake of explicitness, suppose that our array is $A = (a_0, a_1, \ldots, a_{n-1})$. Let's also think of our processors as a linear array: $P = (q_0, q_1, \ldots, q_{p-1})$. We'll assume that the amount of computation associated with each array element is about the same. In other words, if we assign the same number of elements to each processor, we'll have achieved the goal of load balancing.

If the number of processors, p, is equal to the number of array elements, n, then there is only one mapping that balances the load equally among the processors:

$$a_i \to q_i$$

for each i, and our problem seems a trivial one. Indeed, if p evenly divides n, then it might at first seem that there are only two mappings that balance the load. A **block mapping** partitions the array elements into blocks of consecutive entries and assigns the blocks to the processors. Suppose, for example, that $p = 3$ and $n = 12$. Then a block mapping would look like this:

$$a_0, a_1, a_2, a_3 \to q_0,$$
$$a_4, a_5, a_6, a_7 \to q_1,$$
$$a_8, a_9, a_{10}, a_{11} \to q_2.$$

The other "obvious" mapping is a **cyclic mapping**. It assigns the first element to the first processor, the second element to the second, and so on. When each processor has one element of the array, we go back to the first processor, and repeat the assignment process with the next p elements. This process is repeated until all the elements are assigned. If $p = 3$ and $n = 12$, we'll have the following mapping:

$$a_0, a_3, a_6, a_9 \to q_0,$$
$$a_1, a_4, a_7, a_{10} \to q_1,$$
$$a_2, a_5, a_8, a_{11} \to q_2.$$

But we've only scratched the surface! Consider, for example, the following

mapping:

$$a_0, a_1, a_6, a_7 \to q_0,$$
$$a_2, a_3, a_8, a_9 \to q_1,$$
$$a_4, a_5, a_{10}, a_{11} \to q_2.$$

This is a **block-cyclic mapping**. It partitions the array into blocks of consecutive elements as in the block mapping. However, the blocks are not necessarily of size n/p. The blocks are then mapped to the processors in the same way that the elements are mapped in the cyclic mapping. In our example, the blocks have size 2. If we start considering the (very real) possibility that the blocksize and/or p don't evenly divide n, we see that there are a huge number of different mappings just for linear arrays, and if we start looking at higher-dimensional arrays or trees or general graphs, the problem becomes astronomically complex.

So, how do we decide on the appropriate mapping? Not surprisingly, it's highly problem dependent. The literature is filled with discussions of mappings. See the references at the end of the chapter for pointers to information on matrix mappings. We'll come back and look at the nuts and bolts of how to actually distribute an array in section 8.4.1.

2.3 Summary

In this chapter we've touched on a large variety of issues in parallel computing. We began with a discussion of parallel architectures and continued with a discussion of some issues that arise in programming parallel systems.

Pipeline/vector processors obtain parallelism by "pipelining" functional units in the CPU and issuing vector instructions. They continue to be very popular, mainly due to the relative ease with which they can be programmed to solve problems with regular structures. However, they are less successful with irregular structures and don't scale to arbitrarily large problems.

SIMD systems have one control unit and many subordinate arithmetic and logic units. They scale well, and they don't suffer from many of the problems inherent in communicating between asynchronous processes. However, their relative difficulty with irregular structures and their difficulties with conditional branches have led many to believe that they cannot be good general-purpose systems. At this time, it appears that they will probably continue to be niche machines.

The concept of *shared-memory MIMD* is appealing and intuitively natural to programmers accustomed to programming conventional systems. Hence, they have achieved a much wider acceptance than distributed-memory systems. The principal difficulty they have encountered is *scalability*: the hardware needed to allow many processors uniform access to memory is very expensive. The compromise that has been reached is to settle for *nonuniform access*. That is, each processor sees a hierarchy of memory speeds, and it is up to the programmer to keep this in mind when she designs her software. At the top of the hierarchy, most shared-memory systems use fairly large local *caches*. Assuring that these caches are consistent adds to the cost of shared-memory designs.

Distributed-memory MIMD systems continue to scale better than shared-memory systems, and meshes and switch-based systems are currently the most popular architectures. *Routing* is of critical importance in the design of distributed-memory systems; we briefly discussed *store-and-forward* and *cut-through routing*. The principal drawback to distributed-memory MIMD systems has been that they are very difficult to program.

While *processor* and *memory* are the fundamental conceptual units of parallel hardware, the *process* is the fundamental conceptual unit of parallel software. Roughly speaking, a process is an instance of a program that is executing on a physical processor.

Most shared-memory systems provide facilities for both static and dynamic creation of processes. A commonly used method for the dynamic creation/destruction of processes is the familiar `fork/join`. In order to program shared-memory systems, we need programming primitives for defining shared variables—variables that each process can access. We saw that the existence of shared resources can lead to errors if we're not careful to limit access to the shared resources. Sections of code that should only be accessed by one process at a time are called *critical sections*. We used *binary semaphores* to limit access. We may also need to synchronize the processes. A common means for doing this is called a *barrier*.

Message passing is the most commonly used method for programming distributed-memory systems. We saw that message passing can be *synchronous* or *asynchronous*. If a system provides *buffering*, then the system can copy the sender's message to a system buffer, and the sender can continue with its work. However, if there is no buffering, the processes must synchronize; i.e., the sender must receive permission from the receiver to transmit the message. Message-passing functions can also be either *blocking* or *nonblocking*. In blocking message passing, a call to a communication function won't return until the operation is complete. For example, a blocking receive function will not return until the message has been copied into the user process's memory. Nonblocking communication consists of two phases. During the first phase, a function is called that starts the communication. During the second phase, another function is called that completes the communication. Thus, if the system has the capability to simultaneously compute and communicate, we can overlap communication and computation by doing some useful computation between the two phases of the operation.

Parallel programs are usually broadly divided into two categories: *data parallel* and *control parallel*. In data-parallel programs, we obtain parallelism by partitioning the data among the processes; in control-parallel programs, we partition the instructions. Typical parallel programs usually use both methods.