CHAPTER

Parallel Hardware and Parallel Software

2

It's perfectly feasible for specialists in disciplines other than computer science and computer engineering to write parallel programs. However, in order to write *efficient* parallel programs, we do need some knowledge of the underlying hardware and system software. It's also very useful to have some knowledge of different types of parallel software, so in this chapter we'll take a brief look at a few topics in hardware and software. We'll also take a brief look at evaluating program performance and a method for developing parallel programs. We'll close with a discussion of what kind of environment we might expect to be working in, and a few rules and assumptions we'll make in the rest of the book.

This is a long, broad chapter, so it may be a good idea to skim through some of the sections on a first reading so that you have a good idea of what's in the chapter. Then, when a concept or term in a later chapter isn't quite clear, it may be helpful to refer back to this chapter. In particular, you may want to skim over most of the material in "Modifications to the von Neumann Model," except "The Basics of Caching." Also, in the "Parallel Hardware" section, you can safely skim the material on "SIMD Systems" and "Interconnection Networks."

2.1 SOME BACKGROUND

Parallel hardware and software have grown out of conventional **serial** hardware and software: hardware and software that runs (more or less) a single job at a time. So in order to better understand the current state of parallel systems, let's begin with a brief look at a few aspects of serial systems.

2.1.1 The von Neumann architecture

The "classical" **von Neumann architecture** consists of **main memory,** a **central-processing unit** (CPU) or **processor** or **core,** and an **interconnection** between the memory and the CPU. Main memory consists of a collection of locations, each of which is capable of storing both instructions and data. Every location consists of an address, which is used to access the location and the contents of the location—the instructions or data stored in the location.

The central processing unit is divided into a control unit and an arithmetic and logic unit (ALU). The control unit is responsible for deciding which instructions in a program should be executed, and the ALU is responsible for executing the actual instructions. Data in the CPU and information about the state of an executing program are stored in special, very fast storage called **registers**. The control unit has a special register called the **program counter**. It stores the address of the next instruction to be executed.

Instructions and data are transferred between the CPU and memory via the interconnect. This has traditionally been a **bus**, which consists of a collection of parallel wires and some hardware controlling access to the wires. A von Neumann machine executes a single instruction at a time, and each instruction operates on only a few pieces of data. See Figure 2.1.

When data or instructions are transferred from memory to the CPU, we sometimes say the data or instructions are **fetched** or **read** from memory. When data are transferred from the CPU to memory, we sometimes say the data are **written to memory** or **stored**. The separation of memory and CPU is often called the **von Neumann**

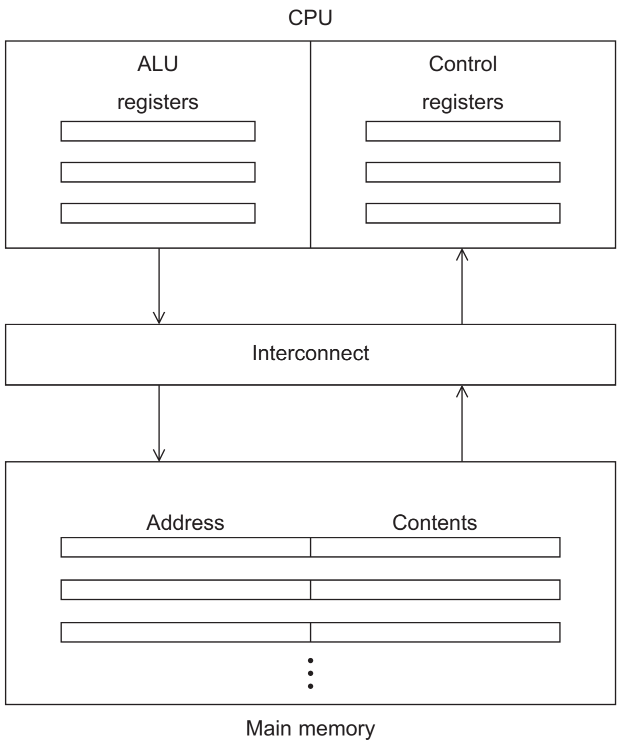

FIGURE 2.1

The von Neumann architecture

bottleneck, since the interconnect determines the rate at which instructions and data can be accessed. The potentially vast quantity of data and instructions needed to run a program is effectively isolated from the CPU. In 2010 CPUs are capable of executing instructions more than one hundred times faster than they can fetch items from main memory.

In order to better understand this problem, imagine that a large company has a single factory (the CPU) in one town and a single warehouse (main memory) in another. Further imagine that there is a single two-lane road joining the warehouse and the factory. All the raw materials used in manufacturing the products are stored in the warehouse. Also, all the finished products are stored in the warehouse before being shipped to customers. If the rate at which products can be manufactured is much larger than the rate at which raw materials and finished products can be transported, then it's likely that there will be a huge traffic jam on the road, and the employees and machinery in the factory will either be idle for extended periods or they will have to reduce the rate at which they produce finished products.

In order to address the von Neumann bottleneck, and, more generally, improve CPU performance, computer engineers and computer scientists have experimented with many modifications to the basic von Neumann architecture. Before discussing some of these modifications, let's first take a moment to discuss some aspects of the software that are used in both von Neumann systems and more modern systems.

2.1.2 Processes, multitasking, and threads

Recall that the **operating system**, or OS, is a major piece of software whose purpose is to manage hardware and software resources on a computer. It determines which programs can run and when they can run. It also controls the allocation of memory to running programs and access to peripheral devices such as hard disks and network interface cards.

When a user runs a program, the operating system creates a **process**—an instance of a computer program that is being executed. A process consists of several entities:

- The executable machine language program.
- A block of memory, which will include the executable code, a **call stack** that keeps track of active functions, a **heap**, and some other memory locations.
- Descriptors of resources that the operating system has allocated to the process—for example, file descriptors.
- Security information—for example, information specifying which hardware and software resources the process can access.
- Information about the state of the process, such as whether the process is ready to run or is waiting on some resource, the content of the registers, and information about the process' memory.

Most modern operating systems are **multitasking**. This means that the operating system provides support for the apparent simultaneous execution of multiple programs. This is possible even on a system with a single core, since each process runs

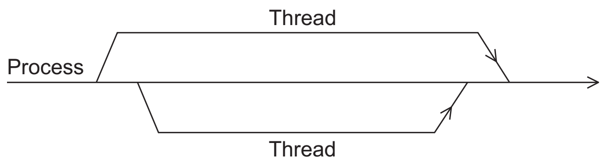

FIGURE 2.2

A process and two threads

for a small interval of time (typically a few milliseconds), often called a **time slice**. After one running program has executed for a time slice, the operating system can run a different program. A multitasking OS may change the running process many times a minute, even though changing the running process can take a long time.

In a multitasking OS if a process needs to wait for a resource—for example, it needs to read data from external storage—it will **block**. This means that it will stop executing and the operating system can run another process. However, many programs can continue to do useful work even though the part of the program that is currently executing must wait on a resource. For example, an airline reservation system that is blocked waiting for a seat map for one user could provide a list of available flights to another user. **Threading** provides a mechanism for programmers to divide their programs into more or less independent tasks with the property that when one thread is blocked another thread can be run. Furthermore, in most systems it's possible to switch between threads much faster than it's possible to switch between processes. This is because threads are "lighter weight" than processes. Threads are contained within processes, so they can use the same executable, and they usually share the same memory and the same I/O devices. In fact, two threads belonging to one process can share most of the process' resources. The two most important exceptions are that they'll need a record of their own program counters and they'll need their own call stacks so that they can execute independently of each other.

If a process is the "master" thread of execution and threads are started and stopped by the process, then we can envision the process and its subsidiary threads as lines: when a thread is started, it **forks** off the process; when a thread terminates, it **joins** the process. See Figure 2.2.

2.2 MODIFICATIONS TO THE VON NEUMANN MODEL

As we noted earlier, since the first electronic digital computers were developed back in the 1940s, computer scientists and computer engineers have made many improvements to the basic von Neumann architecture. Many are targeted at reducing the problem of the von Neumann bottleneck, but many are also targeted at simply making CPUs faster. In this section we'll look at three of these improvements: caching, virtual memory, and low-level parallelism.

2.2.1 The basics of caching

Caching is one of the most widely used methods of addressing the von Neumann bottleneck. To understand the ideas behind caching, recall our example. A company has a factory (CPU) in one town and a warehouse (main memory) in another, and there is a single, two-lane road joining the factory and the warehouse. There are a number of possible solutions to the problem of transporting raw materials and finished products between the warehouse and the factory. One is to widen the road. Another is to move the factory and/or the warehouse or to build a unified factory and warehouse. Caching exploits both of these ideas. Rather than transporting a single instruction or data item, we can use an effectively wider interconnection, an interconnection that can transport more data or more instructions in a single memory access. Also, rather than storing all data and instructions exclusively in main memory, we can store blocks of data and instructions in special memory that is effectively closer to the registers in the CPU.

In general a **cache** is a collection of memory locations that can be accessed in less time than some other memory locations. In our setting, when we talk about caches we'll usually mean a **CPU cache**, which is a collection of memory locations that the CPU can access more quickly than it can access main memory. A CPU cache can either be located on the same chip as the CPU or it can be located on a separate chip that can be accessed much faster than an ordinary memory chip.

Once we have a cache, an obvious problem is deciding which data and instructions should be stored in the cache. The universally used principle is based on the idea that programs tend to use data and instructions that are physically close to recently used data and instructions. After executing an instruction, programs typically execute the next instruction; branching tends to be relatively rare. Similarly, after a program has accessed one memory location, it often accesses a memory location that is physically nearby. An extreme example of this is in the use of arrays. Consider the loop

```
float z[1000];
. . .
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

Arrays are allocated as blocks of contiguous memory locations. So, for example, the location storing `z[1]` immediately follows the location `z[0]`. Thus as long as `i < 999`, the read of `z[i]` is immediately followed by a read of `z[i+1]`.

The principle that an access of one location is followed by an access of a nearby location is often called **locality**. After accessing one memory location (instruction or data), a program will typically access a nearby location (**spatial** locality) in the near future (**temporal** locality).

In order to exploit the principle of locality, the system uses an effectively *wider* interconnect to access data and instructions. That is, a memory access will effectively operate on blocks of data and instructions instead of individual instructions and individual data items. These blocks are called **cache blocks** or **cache lines**. A typical cache line stores 8 to 16 times as much information as a single memory location. In our example, if a cache line stores 16 floats, then when we first go to add `sum += z[0]`, the system might read the first 16 elements of `z`, `z[0]`, `z[1]`, ..., `z[15]` from memory into cache. So the next 15 additions will use elements of `z` that are already in the cache.

Conceptually, it's often convenient to think of a CPU cache as a single monolithic structure. In practice, though, the cache is usually divided into **levels**: the first level (L1) is the smallest and the fastest, and higher levels (L2, L3, ...) are larger and slower. Most systems currently, in 2010, have at least two levels and having three levels is quite common. Caches usually store copies of information in slower memory, and, if we think of a lower-level (faster, smaller) cache as a cache for a higher level, this usually applies. So, for example, a variable stored in a level 1 cache will also be stored in level 2. However, some multilevel caches don't duplicate information that's available in another level. For these caches, a variable in a level 1 cache might not be stored in any other level of the cache, but it *would* be stored in main memory.

When the CPU needs to access an instruction or data, it works its way down the cache hierarchy: First it checks the level 1 cache, then the level 2, and so on. Finally, if the information needed isn't in any of the caches, it accesses main memory. When a cache is checked for information and the information is available, it's called a **cache hit** or just a **hit**. If the information isn't available, it's called a **cache miss** or a **miss**. Hit or miss is often modified by the level. For example, when the CPU attempts to access a variable, it might have an L1 miss and an L2 hit.

Note that the memory access terms **read** and **write** are also used for caches. For example, we might read an instruction from an L2 cache, and we might write data to an L1 cache.

When the CPU attempts to read data or instructions and there's a cache read-miss, it will read from memory the cache line that contains the needed information and store it in the cache. This may stall the processor, while it waits for the slower memory: the processor may stop executing statements from the current program until the required data or instructions have been fetched from memory. So in our example, when we read `z[0]`, the processor may stall while the cache line containing `z[0]` is transferred from memory into the cache.

When the CPU writes data to a cache, the value in the cache and the value in main memory are different or **inconsistent**. There are two basic approaches to dealing with the inconsistency. In **write-through** caches, the line is written to main memory when it is written to the cache. In **write-back** caches, the data isn't written immediately. Rather, the updated data in the cache is marked *dirty,* and when the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

2.2.2 Cache mappings

Another issue in cache design is deciding where lines should be stored. That is, if we fetch a cache line from main memory, where in the cache should it be placed? The answer to this question varies from system to system. At one extreme is a **fully associative** cache, in which a new line can be placed at any location in the cache. At the other extreme is a **direct mapped** cache, in which each cache line has a unique location in the cache to which it will be assigned. Intermediate schemes are called ***n*-way set associative**. In these schemes, each cache line can be placed in one of *n* different locations in the cache. For example, in a two way set associative cache, each line can be mapped to one of two locations.

As an example, suppose our main memory consists of 16 lines with indexes 0–15, and our cache consists of 4 lines with indexes 0–3. In a fully associative cache, line 0 can be assigned to cache location 0, 1, 2, or 3. In a direct mapped cache, we might assign lines by looking at their remainder after division by 4. So lines 0, 4, 8, and 12 would be mapped to cache index 0, lines 1, 5, 9, and 13 would be mapped to cache index 1, and so on. In a two way set associative cache, we might group the cache into two sets: indexes 0 and 1 form one set—set 0—and indexes 2 and 3 form another—set 1. So we could use the remainder of the main memory index modulo 2, and cache line 0 would be mapped to either cache index 0 or cache index 1. See Table 2.1.

When more than one line in memory can be mapped to several different locations in a cache (fully associative and *n*-way set associative), we also need to be able to decide which line should be replaced or **evicted**. In our preceding example, if, for example, line 0 is in location 0 and line 2 is in location 1, where would we store line 4? The most commonly used scheme is called **least recently used**. As the name suggests, the cache has a record of the relative order in which the blocks have been

Table 2.1 Assignments of a 16-line Main Memory to a 4-line Cache

	Cache Location		
Memory Index	*Fully Assoc*	*Direct Mapped*	*2-way*
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

used, and if line 0 were used more recently than line 2, then line 2 would be evicted and replaced by line 4.

2.2.3 Caches and programs: an example

It's important to remember that the workings of the CPU cache are controlled by the system hardware, and we, the programmers, don't directly determine which data and which instructions are in the cache. However, knowing the principle of spatial and temporal locality allows us to have some indirect control over caching. As an example, C stores two-dimensional arrays in "row-major" order. That is, although we think of a two-dimensional array as a rectangular block, memory is effectively a huge one-dimensional array. So in row-major storage, we store row 0 first, then row 1, and so on. In the following two code segments, we would expect the first pair of nested loops to have much better performance than the second, since it's accessing the data in the two-dimensional array in contiguous blocks.

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

To better understand this, suppose `MAX` is four, and the elements of `A` are stored in memory as follows:

Cache Line	**Elements of** `A`			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

So, for example, `A[0][1]` is stored immediately after `A[0][0]` and `A[1][0]` is stored immediately after `A[0][3]`.

Let's suppose that none of the elements of `A` are in the cache when each pair of loops starts executing. Let's also suppose that a cache line consists of four elements of A and A[0][0] is the first element of a cache line. Finally, we'll suppose that the cache is direct mapped and it can only store eight elements of A, or two cache lines. (We won't worry about x and y.)

Both pairs of loops attempt to first access A[0][0]. Since it's not in the cache, this will result in a cache miss, and the system will read the line consisting of the first row of A, A[0][0], A[0][1], A[0][2], A[0][3], into the cache. The first pair of loops then accesses A[0][1], A[0][2], A[0][3], all of which are in the cache, and the next miss in the first pair of loops will occur when the code accesses A[1][0]. Continuing in this fashion, we see that the first pair of loops will result in a total of four misses when it accesses elements of A, one for each row. Note that since our hypothetical cache can only store two lines or eight elements of A, when we read the first element of row two and the first element of row three, one of the lines that's already in the cache will have to be evicted from the cache, but once a line is evicted, the first pair of loops won't need to access the elements of that line again.

After reading the first row into the cache, the second pair of loops needs to then access A[1][0], A[2][0], A[3][0], none of which are in the cache. So the next three accesses of A will also result in misses. Furthermore, because the cache is small, the reads of A[2][0] and A[3][0] will require that lines already in the cache be evicted. Since A[2][0] is stored in cache line 2, reading its line will evict line 0, and reading A[3][0] will evict line 1. After finishing the first pass through the outer loop, we'll next need to access A[0][1], which was evicted with the rest of the first row. So we see that *every* time we read an element of A, we'll have a miss, and the second pair of loops results in 16 misses.

Thus, we'd expect the first pair of nested loops to be much faster than the second. In fact, if we run the code on one of our systems with MAX = 1000, the first pair of nested loops is approximately three times faster than the second pair.

2.2.4 Virtual memory

Caches make it possible for the CPU to quickly access instructions and data that are in main memory. However, if we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory. This is especially true with multitasking operating systems; in order to switch between programs and create the illusion that multiple programs are running simultaneously, the instructions and data that will be used during the next time slice should be in main memory. Thus, in a multitasking system, even if the main memory is very large, many running programs must share the available main memory. Furthermore, this sharing must be done in such a way that each program's data and instructions are protected from corruption by other programs.

Virtual memory was developed so that main memory can function as a cache for secondary storage. It exploits the principle of spatial and temporal locality by keeping in main memory only the active parts of the many running programs; those parts that are idle are kept in a block of secondary storage called **swap space**. Like CPU caches, virtual memory operates on blocks of data and instructions. These blocks are commonly called **pages**, and since secondary storage access can be hundreds of thousands of times slower than main memory access, pages are relatively large—most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.

We may run into trouble if we try to assign physical memory addresses to pages when we compile a program. If we do this, then each page of the program can only be assigned to one block of memory, and with a multitasking operating system, we're likely to have many programs wanting to use the same block of memory. In order to avoid this problem, when a program is compiled, its pages are assigned *virtual* page numbers. Then, when the program is run, a table is created that maps the virtual page numbers to physical addresses. When the program is run and it refers to a virtual address, this **page table** is used to translate the virtual address into a physical address. If the creation of the page table is managed by the operating system, it can ensure that the memory used by one program doesn't overlap the memory used by another.

A drawback to the use of a page table is that it can double the time needed to access a location in main memory. Suppose, for example, that we want to execute an instruction in main memory. Then our executing program will have the *virtual* address of this instruction, but before we can find the instruction in memory, we'll need to translate the virtual address into a physical address. In order to do this, we'll need to find the page in memory that contains the instruction. Now the virtual page number is stored as a part of the virtual address. As an example, suppose our addresses are 32 bits and our pages are 4 kilobytes = 4096 bytes. Then each byte in the page can be identified with 12 bits, since $2^{12} = 4096$. Thus, we can use the low-order 12 bits of the virtual address to locate a byte within a page, and the remaining bits of the virtual address can be used to locate an individual page. See Table 2.2. Observe that the virtual page number can be computed from the virtual address without going to memory. However, once we've found the virtual page number, we'll need to access the page table to translate it into a physical page. If the required part of the page table isn't in cache, we'll need to load it from memory. After it's loaded, we can translate our virtual address to a physical address and get the required instruction.

Table 2.2 Virtual Address Divided into Virtual Page Number and Byte Offset

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	⋯	13	12	11	10	⋯	1	0
1	0	⋯	1	1	0	0	⋯	1	1

This is clearly a problem. Although multiple programs can use main memory at more or less the same time, using a page table has the potential to significantly increase each program's overall run-time. In order to address this issue, processors have a special address translation cache called a **translation-lookaside buffer**, or TLB. It caches a small number of entries (typically 16–512) from the page table in very fast memory. Using the principle of spatial and temporal locality, we would expect that most of our memory references will be to pages whose physical address is stored in the TLB, and the number of memory references that require accesses to the page table in main memory will be substantially reduced.

The terminology for the TLB is the same as the terminology for caches. When we look for an address and the virtual page number is in the TLB, it's called a TLB *hit*. If it's not in the TLB, it's called a *miss*. The terminology for the page table, however, has an important difference from the terminology for caches. If we attempt to access a page that's not in memory, that is, the page table doesn't have a valid physical address for the page and the page is only stored on disk, then the attempted access is called a **page fault**.

The relative slowness of disk accesses has a couple of additional consequences for virtual memory. First, with CPU caches we could handle write-misses with either a write-through or write-back scheme. With virtual memory, however, disk accesses are so expensive that they should be avoided whenever possible, so virtual memory always uses a write-back scheme. This can be handled by keeping a bit on each page in memory that indicates whether the page has been updated. If it has been updated, when it is evicted from main memory, it will be written to disk. Second, since disk accesses are so slow, management of the page table and the handling of disk accesses can be done by the operating system. Thus, even though we as programmers don't directly control virtual memory, unlike CPU caches, which are handled by system hardware, virtual memory is usually controlled by a combination of system hardware and operating system software.

2.2.5 Instruction-level parallelism

Instruction-level parallelism, or ILP, attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions. There are two main approaches to ILP: **pipelining**, in which functional units are arranged in stages, and **multiple issue**, in which multiple instructions can be simultaneously initiated. Both approaches are used in virtually all modern CPUs.

Pipelining

The principle of pipelining is similar to a factory assembly line: while one team is bolting a car's engine to the chassis, another team can connect the transmission to the engine and the driveshaft of a car that's already been processed by the first team, and a third team can bolt the body to the chassis in a car that's been processed by the first two teams. As an example involving computation, suppose we want to add the floating point numbers 9.87×10^4 and 6.54×10^3. Then we can use the following steps:

Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift one operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Here we're using base 10 and a three digit mantissa or significand with one digit to the left of the decimal point. Thus, in the example, normalizing shifts the decimal point one unit to the left, and rounding rounds to three digits.

Now if each of the operations takes one nanosecond (10^{-9} seconds), the addition operation will take seven nanoseconds. So if we execute the code

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

the `for` loop will take something like 7000 nanoseconds.

As an alternative, suppose we divide our floating point adder into seven separate pieces of hardware or functional units. The first unit will fetch two operands, the second will compare exponents, and so on. Also suppose that the output of one functional unit is the input to the next. So, for example, the output of the functional unit that adds the two values is the input to the unit that normalizes the result. Then a single floating point addition will still take seven nanoseconds. However, when we execute the `for` loop, we can fetch `x[1]` and `y[1]` while we're comparing the exponents of `x[0]` and `y[0]`. More generally, it's possible for us to simultaneously execute seven different stages in seven different additions. See Table 2.3. From the table we see that after time 5, the pipelined loop produces a result every nanosecond, instead of every seven nanoseconds, so the total time to execute the `for` loop has been reduced from 7000 nanoseconds to 1006 nanoseconds—an improvement of almost a factor of seven.

In general, a pipeline with k stages won't get a k-fold improvement in performance. For example, if the times required by the various functional units are different, then the stages will effectively run at the speed of the slowest functional unit. Furthermore, delays such as waiting for an operand to become available can cause the pipeline to stall. See Exercise 2.1 for more details on the performance of pipelines.

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Multiple issue

Pipelines improve performance by taking individual pieces of hardware or functional units and connecting them in sequence. Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program. For example, if we have two complete floating point adders, we can approximately halve the time it takes to execute the loop

```
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

While the first adder is computing `z[0]`, the second can compute `z[1]`; while the first is computing `z[2]`, the second can compute `z[3]`; and so on.

If the functional units are scheduled at compile time, the multiple issue system is said to use **static** multiple issue. If they're scheduled at run-time, the system is said to use **dynamic** multiple issue. A processor that supports dynamic multiple issue is sometimes said to be **superscalar**.

Of course, in order to make use of multiple issue, the system must find instructions that can be executed simultaneously. One of the most important techniques is **speculation**. In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess. As a simple example, in the following code, the system might predict that the outcome of `z = x + y` will give `z` a positive value, and, as a consequence, it will assign `w = x`.

```
z = x + y;
if (z > 0)
   w = x;
else
   w = y;
```

As another example, in the code

```
z = x + y;
w = *a_p;  /* a_p is a pointer */
```

the system might predict that `a_p` does not refer to `z`, and hence it can simultaneously execute the two assignments.

As both examples make clear, speculative execution must allow for the possibility that the predicted behavior is incorrect. In the first example, we will need to go back and execute the assignment `w = y` if the assignment `z = x + y` results in a value that's not positive. In the second example, if `a_p` does point to `z`, we'll need to re-execute the assignment `w = *a_p`.

If the compiler does the speculation, it will usually insert code that tests whether the speculation was correct, and, if not, takes corrective action. If the hardware does the speculation, the processor usually stores the result(s) of the speculative execution in a buffer. When it's known that the speculation was correct, the contents of the buffer are transferred to registers or memory. If the speculation was incorrect, the contents of the buffer are discarded and the instruction is re-executed.

While dynamic multiple issue systems can execute instructions out of order, in current generation systems the instructions are still loaded in order and the results of the instructions are also committed in order. That is, the results of instructions are written to registers and memory in the program-specified order.

Optimizing compilers, on the other hand, can reorder instructions. This, as we'll see later, can have important consequences for shared-memory programming.

2.2.6 Hardware multithreading

ILP can be very difficult to exploit: it is a program with a long sequence of dependent statements offers few opportunities. For example, in a direct calculation of the Fibonacci numbers

```
f[0] = f[1] = 1;
for (i = 2; i <= n; i++)
   f[i] = f[i-1] + f[i-2];
```

there's essentially no opportunity for simultaneous execution of instructions.

Thread-level parallelism, or TLP, attempts to provide parallelism through the simultaneous execution of different threads, so it provides a **coarser-grained** parallelism than ILP, that is, the program units that are being simultaneously executed—threads—are larger or coarser than the **finer-grained** units—individual instructions.

Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled—for example, if the current task has to wait for data to be loaded from memory. Instead of looking for parallelism in the currently executing thread, it may make sense to simply run another thread. Of course, in order for this to be useful, the system must support very rapid switching between threads. For example, in some older systems, threads were simply implemented as processes, and in the time it took to switch between processes, thousands of instructions could be executed.

In **fine-grained** multithreading, the processor switches between threads after each instruction, skipping threads that are stalled. While this approach has the potential to avoid wasted machine time due to stalls, it has the drawback that a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction. **Coarse-grained** multithreading attempts to avoid this problem by only switching threads that are stalled waiting for a time-consuming operation to complete (e.g., a load from main memory). This has the virtue that switching threads doesn't need to be nearly instantaneous. However, the processor can be idled on shorter stalls, and thread switching will also cause delays.

Simultaneous multithreading, or SMT, is a variation on fine-grained multithreading. It attempts to exploit superscalar processors by allowing multiple threads to make use of the multiple functional units. If we designate "preferred" threads—threads that have many instructions ready to execute—we can somewhat reduce the problem of thread slowdown.

2.3 PARALLEL HARDWARE

Multiple issue and pipelining can clearly be considered to be parallel hardware, since functional units are replicated. However, since this form of parallelism isn't usually visible to the programmer, we're treating both of them as extensions to the basic von Neumann model, and for our purposes, parallel hardware will be limited to hardware that's visible to the programmer. In other words, if she can readily modify her source code to exploit it, or if she must modify her source code to exploit it, then we'll consider the hardware to be parallel.

2.3.1 SIMD systems

In parallel computing, **Flynn's taxonomy** [18] is frequently used to classify computer architectures. It classifies a system according to the number of instruction streams and the number of data streams it can simultaneously manage. A classical von Neumann system is therefore a **single instruction stream, single data stream**, or SISD system, since it executes a single instruction at a time and it can fetch or store one item of data at a time.

Single instruction, multiple data, or SIMD, systems are parallel systems. As the name suggests, SIMD systems operate on multiple data streams by applying the same instruction to multiple data items, so an abstract SIMD system can be thought of as having a single control unit and multiple ALUs. An instruction is broadcast from the control unit to the ALUs, and each ALU either applies the instruction to the current data item, or it is idle. As an example, suppose we want to carry out a "vector addition." That is, suppose we have two arrays `x` and `y`, each with n elements, and we want to add the elements of `y` to the elements of `x`:

```
for (i = 0; i < n; i++)
   x[i] += y[i];
```

Suppose further that our SIMD system has n ALUs. Then we could load `x[i]` and `y[i]` into the ith ALU, have the ith ALU add `y[i]` to `x[i]`, and store the result in `x[i]`. If the system has m ALUs and $m < n$, we can simply execute the additions in blocks of m elements at a time. For example, if $m = 4$ and $n = 15$, we can first add elements 0 to 3, then elements 4 to 7, then elements 8 to 11, and finally elements 12 to 14. Note that in the last group of elements in our example—elements 12 to 14—we're only operating on three elements of `x` and `y`, so one of the four ALUs will be idle.

The requirement that all the ALUs execute the same instruction or are idle can seriously degrade the overall performance of a SIMD system. For example, suppose we only want to carry out the addition if `y[i]` is positive:

```
for (i = 0; i < n; i++)
   if (y[i] > 0.0) x[i] += y[i];
```

In this setting, we must load each element of `y` into an ALU and determine whether it's positive. If `y[i]` is positive, we can proceed to carry out the addition. Otherwise, the ALU storing `y[i]` will be idle while the other ALUs carry out the addition.

Note also that in a "classical" SIMD system, the ALUs must operate synchronously, that is, each ALU must wait for the next instruction to be broadcast before proceeding. Further, the ALUs have no instruction storage, so an ALU can't delay execution of an instruction by storing it for later execution.

Finally, as our first example shows, SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data. Parallelism that's obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called **data-parallelism**. SIMD parallelism can be very efficient on large data parallel problems, but SIMD systems often don't do very well on other types of parallel problems.

SIMD systems have had a somewhat checkered history. In the early 1990s a maker of SIMD systems (Thinking Machines) was the largest manufacturer of parallel supercomputers. However, by the late 1990s the only widely produced SIMD systems were **vector processors**. More recently, graphics processing units, or GPUs, and desktop CPUs are making use of aspects of SIMD computing.

Vector processors

Although what constitutes a vector processor has changed over the years, their key characteristic is that they can operate on arrays or *vectors* of data, while conventional

CPUs operate on individual data elements or *scalars*. Typical recent systems have the following characteristics:

- *Vector registers*. These are registers capable of storing a vector of operands and operating simultaneously on their contents. The vector length is fixed by the system, and can range from 4 to 128 64-bit elements.
- *Vectorized and pipelined functional units*. Note that the same operation is applied to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors. Thus, vector operations are SIMD.
- *Vector instructions*. These are instructions that operate on vectors rather than scalars. If the vector length is `vector_length`, these instructions have the great virtue that a simple loop such as

  ```
  for (i = 0; i < n; i++)
     x[i] += y[i];
  ```

  requires only a single load, add, and store for each block of `vector_length` elements, while a conventional system requires a load, add, and store for each element.
- *Interleaved memory*. The memory system consists of multiple "banks" of memory, which can be accessed more or less independently. After accessing one bank, there will be a delay before it can be reaccessed, but a different bank can be accessed much sooner. So if the elements of a vector are distributed across multiple banks, there can be little to no delay in loading/storing successive elements.
- *Strided memory access and hardware scatter/gather*. In *strided memory access*, the program accesses elements of a vector located at fixed intervals. For example, accessing the first element, the fifth element, the ninth element, and so on, would be strided access with a stride of four. Scatter/gather (in this context) is writing (scatter) or reading (gather) elements of a vector located at irregular intervals—for example, accessing the first element, the second element, the fourth element, the eighth element, and so on. Typical vector systems provide special hardware to accelerate strided access and scatter/gather.

Vector processors have the virtue that for many applications, they are very fast and very easy to use. Vectorizing compilers are quite good at identifying code that can be vectorized. Further, they identify loops that cannot be vectorized, and they often provide information about why a loop couldn't be vectorized. The user can thereby make informed decisions about whether it's possible to rewrite the loop so that it will vectorize. Vector systems have very high memory bandwidth, and every data item that's loaded is actually used, unlike cache-based systems that may not make use of every item in a cache line. On the other hand, they don't handle irregular data structures as well as other parallel architectures, and there seems to be a very finite limit to their **scalability**, that is, their ability to handle ever larger problems. It's difficult to see how systems could be created that would operate on ever longer vectors. Current generation systems scale by increasing the number of vector processors, not the vector length. Current commodity systems provide limited support for operations on very short vectors, while processors that operate on long vectors are custom manufactured, and, consequently, very expensive.

Graphics processing units

Real-time graphics application programming interfaces, or APIs, use points, lines, and triangles to internally represent the surface of an object. They use a **graphics processing pipeline** to convert the internal representation into an array of pixels that can be sent to a computer screen. Several of the stages of this pipeline are programmable. The behavior of the programmable stages is specified by functions called **shader functions.** The shader functions are typically quite short—often just a few lines of C code. They're also implicitly parallel, since they can be applied to multiple elements (e.g., vertices) in the graphics stream. Since the application of a shader function to nearby elements often results in the same flow of control, GPUs can optimize performance by using SIMD parallelism, and in the current generation all GPUs use SIMD parallelism. This is obtained by including a large number of ALUs (e.g., 80) on each GPU processing core.

Processing a single image can require very large amounts of data—hundreds of megabytes of data for a single image is not unusual. GPUs therefore need to maintain very high rates of data movement, and in order to avoid stalls on memory accesses, they rely heavily on hardware multithreading; some systems are capable of storing the state of more than a hundred suspended threads for each executing thread. The actual number of threads depends on the amount of resources (e.g., registers) needed by the shader function. A drawback here is that many threads processing a lot of data are needed to keep the ALUs busy, and GPUs may have relatively poor performance on small problems.

It should be stressed that GPUs are not pure SIMD systems. Although the ALUs on a given core do use SIMD parallelism, current generation GPUs can have dozens of cores, which are capable of executing independent instruction streams.

GPUs are becoming increasingly popular for general, high-performance computing, and several languages have been developed that allow users to exploit their power. For further details see [30].

2.3.2 MIMD systems

Multiple instruction, multiple data, or MIMD, systems support multiple simultaneous instruction streams operating on multiple data streams. Thus, MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU. Furthermore, unlike SIMD systems, MIMD systems are usually **asynchronous**, that is, the processors can operate at their own pace. In many MIMD systems there is no global clock, and there may be no relation between the system times on two different processors. In fact, unless the programmer imposes some synchronization, even if the processors are executing exactly the same sequence of instructions, at any given instant they may be executing different statements.

As we noted in Chapter 1, there are two principal types of MIMD systems: shared-memory systems and distributed-memory systems. In a **shared-memory system** a collection of autonomous processors is connected to a memory system via an interconnection network, and each processor can access each memory location. In a shared-memory system, the processors usually communicate implicitly by accessing shared data structures. In a **distributed-memory system**, each processor is paired with its own *private* memory, and the processor-memory pairs communicate over an interconnection network. So in distributed-memory systems the processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor. See Figures 2.3 and 2.4.

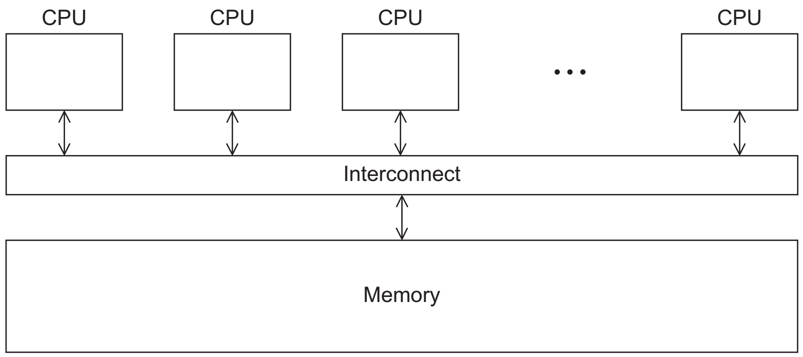

FIGURE 2.3

A shared-memory system

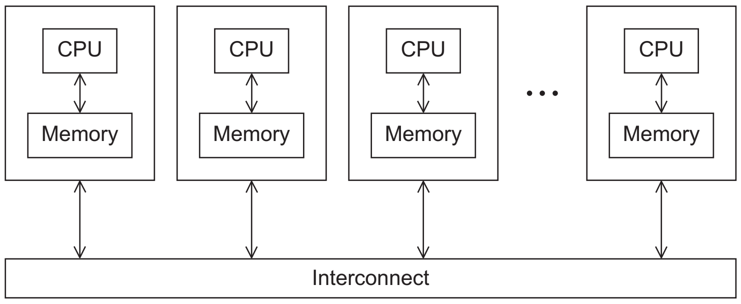

FIGURE 2.4

A distributed-memory system

Shared-memory systems

The most widely available shared-memory systems use one or more **multicore** processors. As we discussed in Chapter 1, a multicore processor has multiple CPUs or cores on a single chip. Typically, the cores have private level 1 caches, while other caches may or may not be shared between the cores.

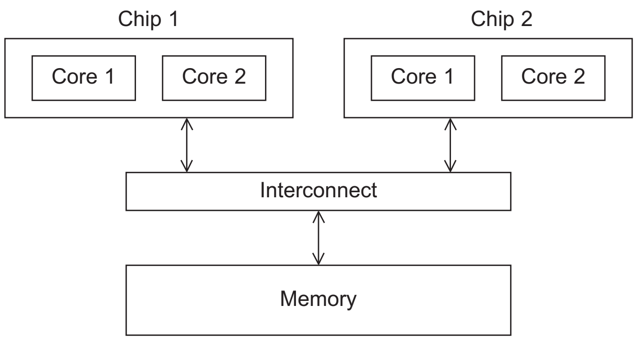

FIGURE 2.5

A UMA multicore system

In shared-memory systems with multiple multicore processors, the interconnect can either connect all the processors directly to main memory or each processor can have a direct connection to a block of main memory, and the processors can access each others' blocks of main memory through special hardware built into the processors. See Figures 2.5 and 2.6. In the first type of system, the time to access all the memory locations will be the same for all the cores, while in the second type a memory location to which a core is directly connected can be accessed more quickly than a memory location that must be accessed through another chip. Thus, the first type of system is called a **uniform memory access**, or UMA, system, while the second type is called a **nonuniform memory access**, or NUMA, system. UMA systems are usually easier to program, since the programmer doesn't need to worry about different access times for different memory locations. This advantage can be offset by the faster access to the directly connected memory in NUMA systems. Furthermore, NUMA systems have the potential to use larger amounts of memory than UMA systems.

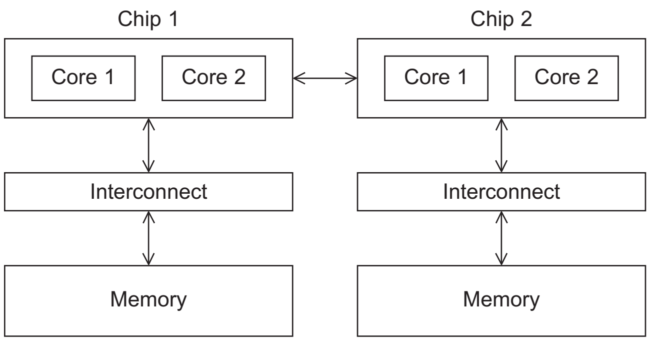

FIGURE 2.6

A NUMA multicore system

Distributed-memory systems

The most widely available distributed-memory systems are called **clusters**. They are composed of a collection of commodity systems—for example, PCs—connected by a commodity interconnection network—for example, Ethernet. In fact, the **nodes** of these systems, the individual computational units joined together by the communication network, are usually shared-memory systems with one or more multicore processors. To distinguish such systems from pure distributed-memory systems, they are sometimes called **hybrid systems**. Nowadays, it's usually understood that a cluster will have shared-memory nodes.

The **grid** provides the infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system. In general, such a system will be *heterogeneous*, that is, the individual nodes may be built from different types of hardware.

2.3.3 Interconnection networks

The interconnect plays a decisive role in the performance of both distributed- and shared-memory systems: even if the processors and memory have virtually unlimited performance, a slow interconnect will seriously degrade the overall performance of all but the simplest parallel program. See, for example, Exercise 2.10.

Although some of the interconnects have a great deal in common, there are enough differences to make it worthwhile to treat interconnects for shared-memory and distributed-memory separately.

Shared-memory interconnects

Currently the two most widely used interconnects on shared-memory systems are buses and crossbars. Recall that a **bus** is a collection of parallel communication wires together with some hardware that controls access to the bus. The key characteristic of a bus is that the communication wires are shared by the devices that are connected to it. Buses have the virtue of low cost and flexibility; multiple devices can be connected to a bus with little additional cost. However, since the communication wires are shared, as the number of devices connected to the bus increases, the likelihood that there will be contention for use of the bus increases, and the expected performance of the bus decreases. Therefore, if we connect a large number of processors to a bus, we would expect that the processors would frequently have to wait for access to main memory. Thus, as the size of shared-memory systems increases, buses are rapidly being replaced by *switched* interconnects.

As the name suggests, **switched** interconnects use switches to control the routing of data among the connected devices. A **crossbar** is illustrated in Figure 2.7(a). The lines are bidirectional communication links, the squares are cores or memory modules, and the circles are switches.

The individual switches can assume one of the two configurations shown in Figure 2.7(b). With these switches and at least as many memory modules as processors, there will only be a conflict between two cores attempting to access memory

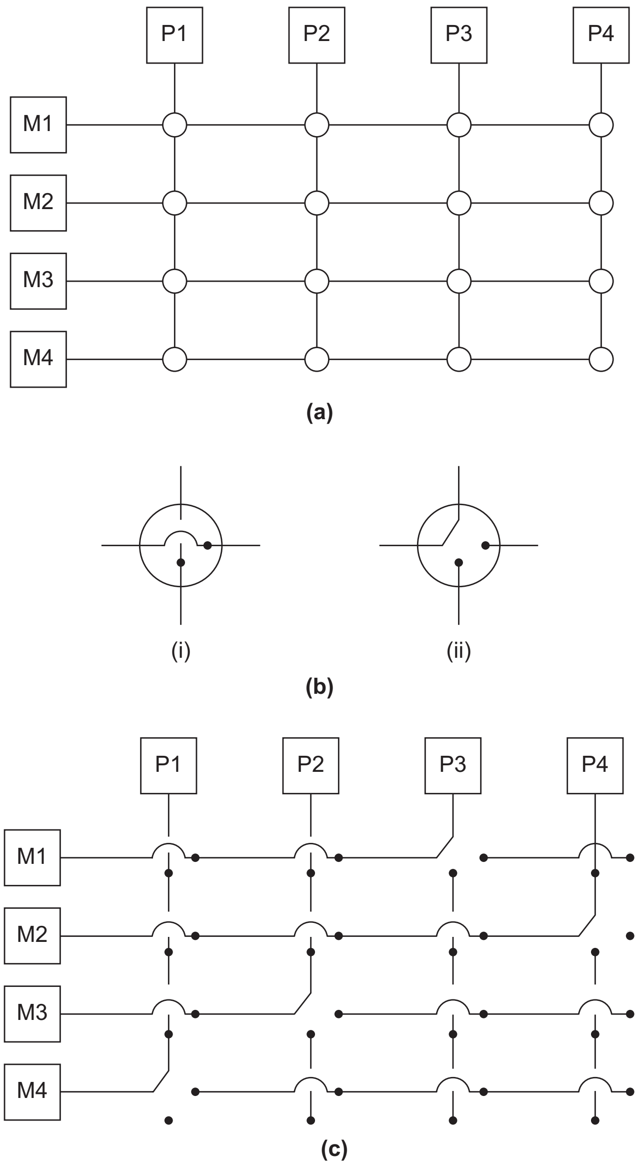

FIGURE 2.7

(a) A crossbar switch connecting four processors (P_i) and four memory modules (M_j); (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors if the two cores attempt to simultaneously access the same memory module. For example, Figure 2.7(c) shows the configuration of the switches if P_1 writes to M_4, P_2 reads from M_3, P_3 reads from M_1, and P_4 writes to M_2.

Crossbars allow simultaneous communication among different devices, so they are much faster than buses. However, the cost of the switches and links is relatively high. A small bus-based system will be much less expensive than a crossbar-based system of the same size.

Distributed-memory interconnects

Distributed-memory interconnects are often divided into two groups: direct interconnects and indirect interconnects. In a **direct interconnect** each switch is directly connected to a processor-memory pair, and the switches are connected to each other. Figure 2.8 shows a **ring** and a two-dimensional **toroidal mesh**. As before, the circles are switches, the squares are processors, and the lines are bidirectional links. A ring is superior to a simple bus since it allows multiple simultaneous communications. However, it's easy to devise communication schemes in which some of the processors must wait for other processors to complete their communications. The toroidal mesh will be more expensive than the ring, because the switches are more complex—they must support five links instead of three—and if there are p processors, the number of links is $2p$ in a toroidal mesh, while it's only p in a ring. However, it's not difficult to convince yourself that the number of possible simultaneous communications patterns is greater with a mesh than with a ring.

One measure of "number of simultaneous communications" or "connectivity" is **bisection width**. To understand this measure, imagine that the parallel system is

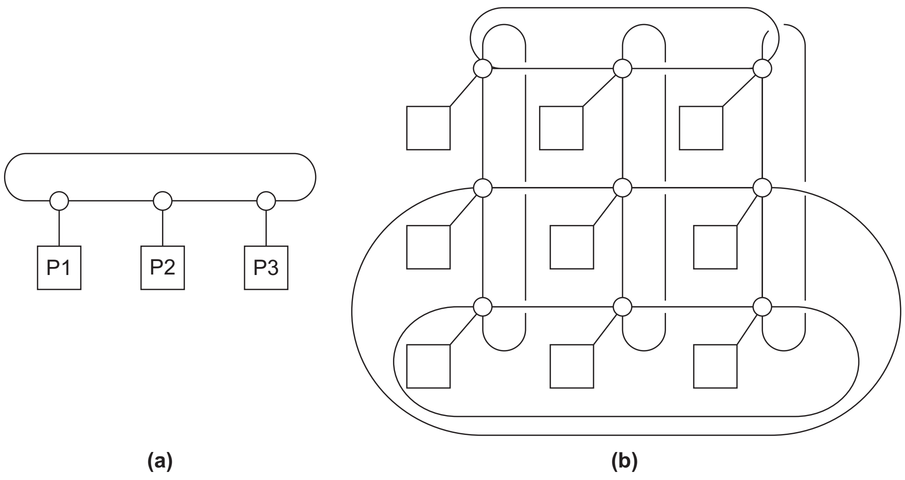

FIGURE 2.8

(a) A ring and (b) a toroidal mesh

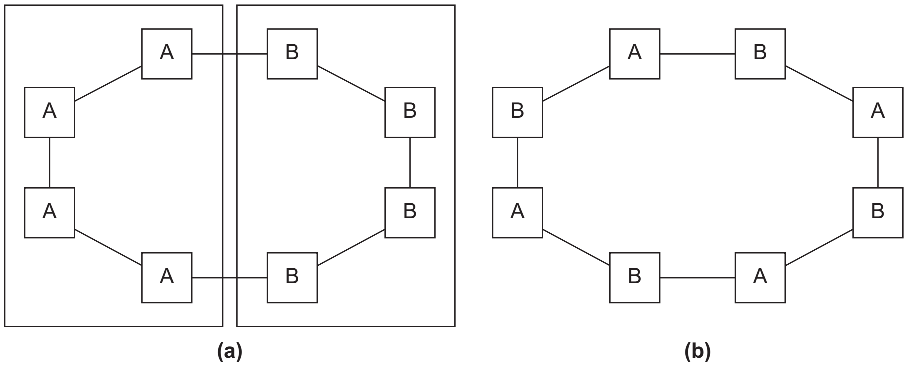

FIGURE 2.9

Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place

divided into two halves, and each half contains half of the processors or nodes. How many simultaneous communications can take place "across the divide" between the halves? In Figure 2.9(a) we've divided a ring with eight nodes into two groups of four nodes, and we can see that only two communications can take place between the halves. (To make the diagrams easier to read, we've grouped each node with its switch in this and subsequent diagrams of direct interconnects.) However, in Figure 2.9(b) we've divided the nodes into two parts so that four simultaneous communications can take place, so what's the bisection width? The bisection width is supposed to give a "worst-case" estimate, so the bisection width is two—not four.

An alternative way of computing the bisection width is to remove the minimum number of links needed to split the set of nodes into two equal halves. The number of links removed is the bisection width. If we have a square two-dimensional toroidal mesh with $p = q^2$ nodes (where q is even), then we can split the nodes into two halves by removing the "middle" horizontal links and the "wraparound" horizontal links. See Figure 2.10. This suggests that the bisection width is at most $2q = 2\sqrt{p}$. In fact, this is the smallest possible number of links and the bisection width of a square two-dimensional toroidal mesh is $2\sqrt{p}$.

The **bandwidth** of a link is the rate at which it can transmit data. It's usually given in megabits or megabytes per second. **Bisection bandwidth** is often used as a measure of network quality. It's similar to bisection width. However, instead of counting the number of links joining the halves, it sums the bandwidth of the links. For example, if the links in a ring have a bandwidth of one billion bits per second, then the bisection bandwidth of the ring will be two billion bits per second or 2000 megabits per second.

The ideal direct interconnect is a **fully connected network** in which each switch is directly connected to every other switch. See Figure 2.11. Its bisection width is $p^2/4$. However, it's impractical to construct such an interconnect for systems with more than a few nodes, since it requires a total of $p^2/2 - p/2$ links, and each switch

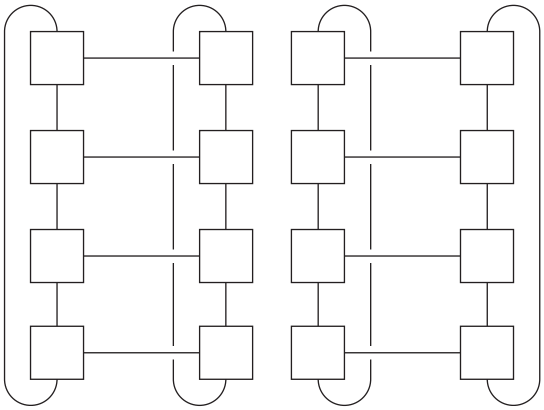

FIGURE 2.10

A bisection of a toroidal mesh

must be capable of connecting to p links. It is therefore more a "theoretical best possible" interconnect than a practical one, and it is used as a basis for evaluating other interconnects.

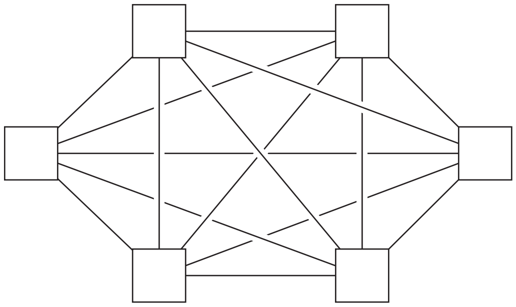

FIGURE 2.11

A fully connected network

The **hypercube** is a highly connected direct interconnect that has been used in actual systems. Hypercubes are built inductively: A one-dimensional hypercube is a fully-connected system with two processors. A two-dimensional hypercube is built from two one-dimensional hypercubes by joining "corresponding" switches. Similarly, a three-dimensional hypercube is built from two two-dimensional hypercubes. See Figure 2.12. Thus, a hypercube of dimension d has $p = 2^d$ nodes, and a switch in a d-dimensional hypercube is directly connected to a processor and d switches. The bisection width of a hypercube is $p/2$, so it has more connectivity than a ring or toroidal mesh, but the switches must be more powerful, since they must support $1 + d = 1 + \log_2(p)$ wires, while the mesh switches only require five wires. So a hypercube with p nodes is more expensive to construct than a toroidal mesh.

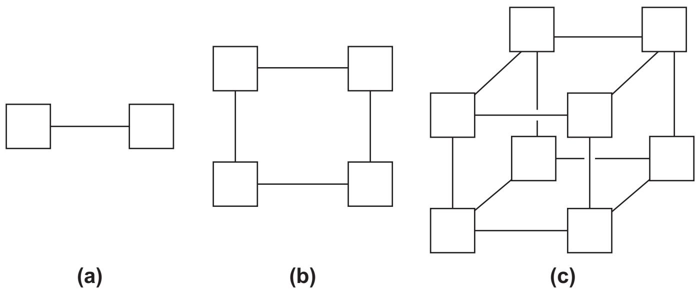

FIGURE 2.12

(a) One-, (b) two-, and (c) three-dimensional hypercubes

Indirect interconnects provide an alternative to direct interconnects. In an indirect interconnect, the switches may not be directly connected to a processor. They're often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network. See Figure 2.13.

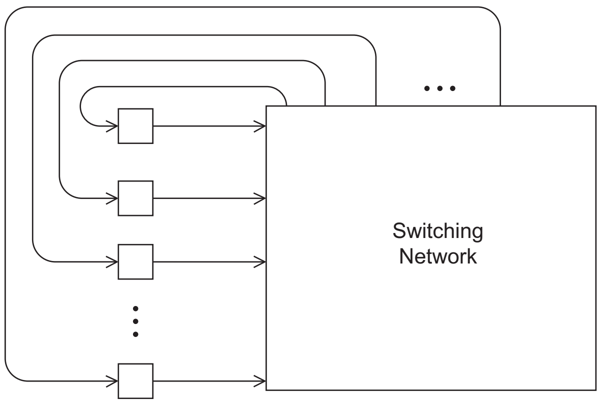

FIGURE 2.13

A generic indirect network

The **crossbar** and the **omega network** are relatively simple examples of indirect networks. We saw a shared-memory crossbar with bidirectional links earlier (Figure 2.7). The diagram of a distributed-memory crossbar in Figure 2.14 has unidirectional links. Notice that as long as two processors don't attempt to communicate with the same processor, all the processors can simultaneously communicate with another processor.

An omega network is shown in Figure 2.15. The switches are two-by-two crossbars (see Figure 2.16). Observe that unlike the crossbar, there are communications that cannot occur simultaneously. For example, in Figure 2.15 if processor 0 sends

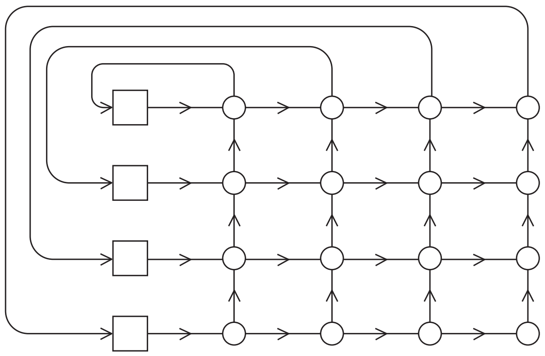

FIGURE 2.14

A crossbar interconnect for distributed-memory

a message to processor 6, then processor 1 cannot simultaneously send a message to processor 7. On the other hand, the omega network is less expensive than the crossbar. The omega network uses $\frac{1}{2}p\log_2(p)$ of the 2×2 crossbar switches, so it uses a total of $2p\log_2(p)$ switches, while the crossbar uses p^2.

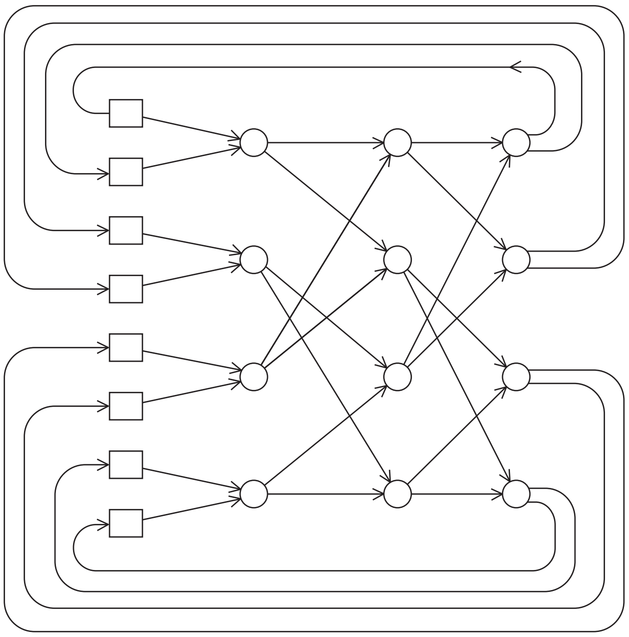

FIGURE 2.15

An omega network

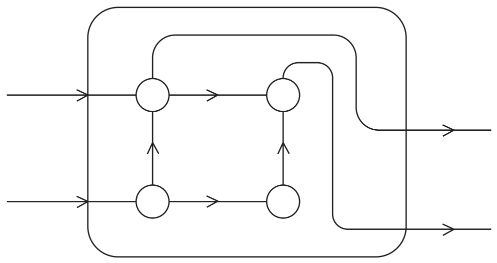

FIGURE 2.16

A switch in an omega network

It's a little bit more complicated to define bisection width for indirect networks. See Exercise 2.14. However, the principle is the same: we want to divide the nodes into two groups of equal size and determine how much communication can take place between the two halves, or alternatively, the minimum number of links that need to be removed so that the two groups can't communicate. The bisection width of a $p \times p$ crossbar is p and the bisection width of an omega network is $p/2$.

Latency and bandwidth

Any time data is transmitted, we're interested in how long it will take for the data to reach its destination. This is true whether we're talking about transmitting data between main memory and cache, cache and register, hard disk and memory, or between two nodes in a distributed-memory or hybrid system. There are two figures that are often used to describe the performance of an interconnect (regardless of what it's connecting): the **latency** and the **bandwidth**. The latency is the time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte. The bandwidth is the rate at which the destination receives data after it has started to receive the first byte. So if the latency of an interconnect is l seconds and the bandwidth is b bytes per second, then the time it takes to transmit a message of n bytes is

$$\text{message transmission time} = l + n/b.$$

Beware, however, that these terms are often used in different ways. For example, latency is sometimes used to describe total message transmission time. It's also often used to describe the time required for any fixed overhead involved in transmitting data. For example, if we're sending a message between two nodes in a distributed-memory system, a message is not just raw data. It might include the data to be transmitted, a destination address, some information specifying the size of the message, some information for error correction, and so on. So in this setting, latency might be the time it takes to assemble the message on the sending side—the time needed to combine the various parts—and the time to disassemble the message on the receiving side—the time needed to extract the raw data from the message and store it in its destination.

2.3.4 Cache coherence

Recall that CPU caches are managed by system hardware: programmers don't have direct control over them. This has several important consequences for shared-memory systems. To understand these issues, suppose we have a shared-memory system with two cores, each of which has its own private data cache. See Figure 2.17. As long as the two cores only read shared data, there is no problem. For example, suppose that `x` is a shared variable that has been initialized to 2, `y0` is private and owned by core 0, and `y1` and `z1` are private and owned by core 1. Now suppose the following statements are executed at the indicated times:

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving `x`
2	Statement(s) not involving `x`	`z1 = 4*x;`

Then the memory location for `y0` will eventually get the value 2, and the memory location for `y1` will eventually get the value 6. However, it's not so clear what value `z1` will get. It might at first appear that since core 0 updates `x` to 7 before the assignment to `z1`, `z1` will get the value $4 \times 7 = 28$. However, at time 0, `x` is in the cache of core 1. So unless for some reason `x` is evicted from core 0's cache and then reloaded into core 1's cache, it actually appears that the original value `x = 2` may be used, and `z1` will get the value $4 \times 2 = 8$.

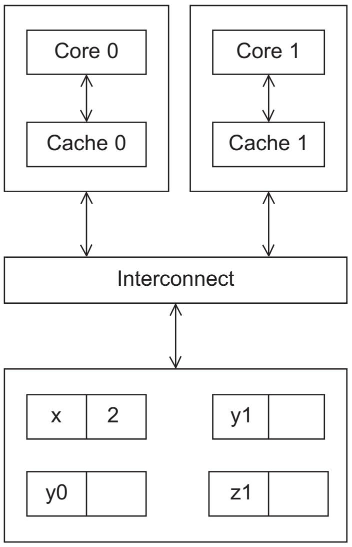

FIGURE 2.17

A shared-memory system with two cores and two caches

Note that this unpredictable behavior will occur regardless of whether the system is using a write-through or a write-back policy. If it's using a write-through policy, the main memory will be updated by the assignment `x = 7`. However, this will have no effect on the value in the cache of core 1. If the system is using a write-back policy, the new value of `x` in the cache of core 0 probably won't even be available to core 1 when it updates `z1`.

Clearly, this is a problem. The programmer doesn't have direct control over when the caches are updated, so her program cannot execute these apparently innocuous statements and know what will be stored in `z1`. There are several problems here, but the one we want to look at right now is that the caches we described for single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cached variable is "seen" by the other processors. That is, that the cached value stored by the other processors is also updated. This is called the **cache coherence** problem.

Snooping cache coherence

There are two main approaches to insuring cache coherence: **snooping cache coherence** and directory-based cache coherence. The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus. Thus, when core 0 updates the copy of `x` stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that `x` has been updated and it can mark its copy of `x` as invalid. This is more or less how snooping cache coherence works. The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the *cache line* containing `x` has been updated, not that `x` has been updated.

A couple of points should be made regarding snooping. First, it's not essential that the interconnect be a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and write-back caches. In principle, if the interconnect is shared—as with a bus—with write-through caches there's no need for additional traffic on the interconnect, since each core can simply "watch" for writes. With write-back caches, on the other hand, an extra communication *is* necessary, since updates to the cache don't get immediately sent to memory.

Directory-based cache coherence

Unfortunately, in large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated (but see Exercise 2.15). So snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade. For example, suppose we have a system with the basic distributed-memory architecture (Figure 2.4). However, the system provides a single address space for all the memories. So, for example, core 0 can access the variable `x` stored in core 1's memory, by simply executing a statement such as `y = x`.

(Of course, accessing the memory attached to another core will be slower than accessing "local" memory, but that's another story.) Such a system can, in principle, scale to very large numbers of cores. However, snooping cache coherence is clearly a problem since a broadcast across the interconnect will be very slow relative to the speed of accessing local memory.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory**. The directory stores the status of each cache line. Typically, this data structure is distributed; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. Thus, when a line is read into, say, core 0's cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Clearly there will be substantial additional storage required for the directory, but when a cache variable is updated, only the cores storing that variable need to be contacted.

False sharing

It's important to remember that CPU caches are implemented in hardware, so they operate on cache lines, not individual variables. This can have disastrous consequences for performance. As an example, suppose we want to repeatedly call a function `f(i,j)` and add the computed values into a vector:

```
int i, j, m, n;
double y[m];

/* Assign y = 0 */
. . .

for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

We can parallelize this by dividing the iterations in the outer loop among the cores. If we have `core_count` cores, we might assign the first `m/core_count` iterations to the first core, the next `m/core_count` iterations to the second core, and so on.

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count
double y[m];

iter_count = m/core_count

/* Core 0 does this */
```

```
for (i = 0; i < iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count; i < 2*iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

. . .
```

Now suppose our shared-memory system has two cores, `m = 8`, doubles are eight bytes, cache lines are 64 bytes, and `y[0]` is stored at the beginning of a cache line. A cache line can store eight doubles, and `y` takes one full cache line. What happens when core 0 and core 1 simultaneously execute their codes? Since all of `y` is stored in a single cache line, each time one of the cores executes the statement `y[i] += f(i,j)`, the line will be invalidated, and the next time the other core tries to execute this statement it will have to fetch the updated line from memory! So if *n* is large, we would expect that a large percentage of the assignments `y[i] += f(i,j)` will access main memory—in spite of the fact that core 0 and core 1 never access each others' elements of `y`. This is called **false sharing**, because the system is behaving *as if* the elements of `y` were being shared by the cores.

Note that false sharing does not cause incorrect results. However, it can ruin the performance of a program by causing many more accesses to memory than necessary. We can reduce its effect by using temporary storage that is local to the thread or process and then copying the temporary storage to the shared storage. We'll return to the subject of false sharing in Chapters 4 and 5.

2.3.5 Shared-memory versus distributed-memory

Newcomers to parallel computing sometimes wonder why all MIMD systems aren't shared-memory, since most programmers find the concept of implicitly coordinating the work of the processors through shared data structures more appealing than explicitly sending messages. There are several issues, some of which we'll discuss when we talk about software for distributed- and shared-memory. However, the principal hardware issue is the cost of scaling the interconnect. As we add processors to a bus, the chance that there will be conflicts over access to the bus increase dramatically, so buses are suitable for systems with only a few processors. Large crossbars are very expensive, so it's also unusual to find systems with large crossbar interconnects. On the other hand, distributed-memory interconnects such as the hypercube and the toroidal mesh are relatively inexpensive, and distributed-memory systems with thousands of processors that use these and other interconnects have been built. Thus, distributed-memory systems are often better suited for problems requiring vast amounts of data or computation.

2.4 PARALLEL SOFTWARE

Parallel hardware has arrived. Virtually all desktop and server systems use multicore processors. The same cannot be said for parallel software. Except for operating systems, database systems, and Web servers, there is currently very little commodity software that makes extensive use of parallel hardware. As we noted in Chapter 1, this is a problem because we can no longer rely on hardware and compilers to provide a steady increase in application performance. If we're to continue to have routine increases in application performance and application power, software developers must learn to write applications that exploit shared- and distributed-memory architectures. In this section we'll take a look at some of the issues involved in writing software for parallel systems.

First, some terminology. Typically when we run our shared-memory programs, we'll start a single process and fork multiple threads. So when we discuss shared-memory programs, we'll talk about *threads* carrying out tasks. On the other hand, when we run distributed-memory programs, we'll start multiple processes, and we'll talk about *processes* carrying out tasks. When the discussion applies equally well to shared-memory and distributed-memory systems, we'll talk about *processes/threads* carrying out tasks.

2.4.1 Caveats

Before proceeding, we need to stress some of the limitations of this section. First, here, and in the remainder of the book, we'll only be discussing software for MIMD systems. For example, while the use of GPUs as a platform for parallel computing continues to grow at a rapid pace, the application programming interfaces (APIs) for GPUs are necessarily very different from standard MIMD APIs. Second, we stress that our coverage is only meant to give some idea of the issues: there is no attempt to be comprehensive.

Finally, we'll mainly focus on what's often called **single program, multiple data**, or SPMD, programs. Instead of running a different program on each core, SPMD programs consist of a single executable that can behave as if it were multiple different programs through the use of conditional branches. For example,

```
if (I'm thread/process 0)
   do this;
else
   do that;
```

Observe that SPMD programs can readily implement data-parallelism. For example,

```
if (I'm thread/process 0)
   operate on the first half of the array;
else /* I'm thread/process 1 */
   operate on the second half of the array;
```

Recall that a program is **task parallel** if it obtains its parallelism by dividing tasks among the threads or processes. The first example makes it clear that SPMD programs can also implement **task-parallelism**.

2.4.2 Coordinating the processes/threads

In a very few cases, obtaining excellent parallel performance is trivial. For example, suppose we have two arrays and we want to add them:

```
double x[n], y[n];
. . .
for (int i = 0; i < n; i++)
   x[i] += y[i];
```

In order to parallelize this, we only need to assign elements of the arrays to the processes/threads. For example, if we have p processes/threads, we might make process/thread 0 responsible for elements $0,\ldots,n/p-1$, process/thread 1 would be responsible for elements $n/p,\ldots,2n/p-1$, and so on.

So for this example, the programmer only needs to

1. Divide the work among the processes/threads
   a. in such a way that each process/thread gets roughly the same amount of work, and
   b. in such a way that the amount of communication required is minimized.

Recall that the process of dividing the work among the processes/threads so that (a) is satisfied is called **load balancing**. The two qualifications on dividing the work are obvious, but nonetheless important. In many cases it won't be necessary to give much thought to them; they typically become concerns in situations in which the amount of work isn't known in advance by the programmer, but rather the work is generated as the program runs. For an example, see the tree search problem in Chapter 6.

Although we might wish for a term that's a little easier to pronounce, recall that the process of converting a serial program or algorithm into a parallel program is often called **parallelization**. Programs that can be parallelized by simply dividing the work among the processes/threads are sometimes said to be **embarrassingly parallel**. This is a bit unfortunate, since it suggests that programmers should be embarrassed to have written an embarrassingly parallel program, when, to the contrary, successfully devising a parallel solution to *any* problem is a cause for great rejoicing.

Alas, the vast majority of problems are much more determined to resist our efforts to find a parallel solution. As we saw in Chapter 1, for these problems, we need to coordinate the work of the processes/threads. In these programs, we also usually need to

2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among the processes/threads.

These last two problems are often interrelated. For example, in distributed-memory programs, we often implicitly synchronize the processes by communicating among them, and in shared-memory programs, we often communicate among the threads by synchronizing them. We'll say more about both issues below.

2.4.3 Shared-memory

As we noted earlier, in shared-memory programs, variables can be **shared** or **private**. Shared variables can be read or written by any thread, and private variables can ordinarily only be accessed by one thread. Communication among the threads is usually done through shared variables, so communication is implicit, rather than explicit.

Dynamic and static threads

In many environments shared-memory programs use **dynamic threads**. In this paradigm, there is often a master thread and at any given instant a (possibly empty) collection of worker threads. The master thread typically waits for work requests—for example, over a network—and when a new request arrives, it forks a worker thread, the thread carries out the request, and when the thread completes the work, it terminates and joins the master thread. This paradigm makes efficient use of system resources since the resources required by a thread are only being used while the thread is actually running.

An alternative to the dynamic paradigm is the **static thread** paradigm. In this paradigm, all of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed. After the threads join the master thread, the master thread may do some cleanup (e.g., free memory) and then it also terminates. In terms of resource usage, this may be less efficient: if a thread is idle, its resources (e.g., stack, program counter, and so on.) can't be freed. However, forking and joining threads can be fairly time-consuming operations. So if the necessary resources are available, the static thread paradigm has the potential for better performance than the dynamic paradigm. It also has the virtue that it's closer to the most widely used paradigm for distributed-memory programming, so part of the mindset that is used for one type of system is preserved for the other. Hence, we'll often use the static thread paradigm.

Nondeterminism

In any MIMD system in which the processors execute asynchronously it is likely that there will be **nondeterminism**. A computation is nondeterministic if a given input can result in different outputs. If multiple threads are executing independently, the relative rate at which they'll complete statements varies from run to run, and hence the results of the program may be different from run to run. As a very simple example, suppose we have two threads, one with id or rank 0 and the other with id or rank 1. Suppose also that each is storing a private variable `my_x`, thread 0's value for `my_x` is 7, and thread 1's is 19. Further, suppose both threads execute the following code:

```
. . .
printf("Thread %d > my_val = %d\n", my_rank, my_x);
. . .
```

Then the output could be

```
Thread 0 > my_val = 7
Thread 1 > my_val = 19
```

but it could also be

```
Thread 1 > my_val = 19
Thread 0 > my_val = 7
```

In fact, things could be even worse: the output of one thread could be broken up by the output of the other thread. However, the point here is that because the threads are executing independently and interacting with the operating system, the time it takes for one thread to complete a block of statements varies from execution to execution, so the order in which these statements complete can't be predicted.

In many cases nondeterminism isn't a problem. In our example, since we've labelled the output with the thread's rank, the order in which the output appears probably doesn't matter. However, there are also many cases in which nondeterminism—especially in shared-memory programs—can be disastrous, because it can easily result in program errors. Here's a simple example with two threads.

Suppose each thread computes an `int`, which it stores in a private variable `my_val`. Suppose also that we want to add the values stored in `my_val` into a shared-memory location `x` that has been initialized to 0. Both threads therefore want to execute code that looks something like this:

```
my_val = Compute_val(my_rank);
x += my_val;
```

Now recall that an addition typically requires loading the two values to be added into registers, adding the values, and finally storing the result. To keep things relatively simple, we'll assume that values are loaded from main memory directly into registers and stored in main memory directly from registers. Here is one possible sequence of events:

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

Clearly this is not what we want, and it's easy to imagine other sequences of events that result in an incorrect value for `x`. The nondeterminism here is a result of the fact that two threads are attempting to more or less simultaneously update the memory location `x`. When threads or processes attempt to simultaneously access a resource, and the accesses can result in an error, we often say the program has a **race condition**, because the threads or processes are in a "horse race." That is, the outcome of the computation depends on which thread wins the race. In our example, the threads are in a race to execute `x += my_val`. In this case, unless one thread completes `x += my_val` before the other thread starts, the result will be incorrect. A block of code that can only be executed by one thread at a time is called a **critical section**, and it's usually our job as programmers to insure **mutually exclusive** access to the critical section. In other words, we need to insure that if one thread is executing the code in the critical section, then the other threads are excluded.

The most commonly used mechanism for insuring mutual exclusion is a **mutual exclusion lock** or **mutex** or **lock**. A mutex is a special type of object that has support in the underlying hardware. The basic idea is that each critical section is *protected* by a lock. Before a thread can execute the code in the critical section, it must "obtain" the mutex by calling a mutex function, and, when it's done executing the code in the critical section, it should "relinquish" the mutex by calling an unlock function. While one thread "owns" the lock—that is, has returned from a call to the lock function, but hasn't yet called the unlock function—any other thread attempting to execute the code in the critical section will wait in its call to the lock function.

Thus, in order to insure that our code functions correctly, we might modify it so that it looks something like this:

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

This insures that only one thread at a time can execute the statement `x += my_val`. Note that the code does *not* impose any predetermined order on the threads. Either thread 0 or thread 1 can execute `x += my_val` first.

Also note that the use of a mutex enforces **serialization** of the critical section. Since only one thread at a time can execute the code in the critical section, this code is effectively serial. Thus, we want our code to have as few critical sections as possible, and we want our critical sections to be as short as possible.

There are alternatives to mutexes. In **busy-waiting**, a thread enters a loop whose sole purpose is to test a condition. In our example, suppose there is a shared variable `ok_for_1` that has been initialized to false. Then something like the following code can insure that thread 1 won't update `x` until after thread 0 has updated it:

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
   while (!ok_for_1);   /* Busy−wait loop */
x += my_val;            /* Critical section */
if (my_rank == 0)
   ok_for_1 = true;     /* Let thread 1 update x */
```

So until thread 0 executes `ok_for_1 = true`, thread 1 will be stuck in the loop `while (!ok_for_1)`. This loop is called a "busy-wait" because the thread can be very busy waiting for the condition. This has the virtue that it's simple to understand and implement. However, it can be very wasteful of system resources, because even when a thread is doing no useful work, the core running the thread will be repeatedly checking to see if the critical section can be entered. **Semaphores** are similar to mutexes, although the details of their behavior are slightly different, and there are some types of thread synchronization that are easier to implement with semaphores than mutexes. A **monitor** provides mutual exclusion at a somewhat higher-level: it is an object whose methods can only be executed by one thread at a time. We'll discuss busy-waiting and semaphores in Chapter 4.

There are a number of other alternatives that are currently being studied but that are not yet widely available. The one that has attracted the most attention is probably **transactional memory** [31]. In database management systems, a **transaction** is an access to a database that the system treats as a single unit. For example, transferring $1000 from your savings account to your checking account should be treated by your bank's software as a transaction, so that the software can't debit your savings account without also crediting your checking account. If the software was able to debit your savings account, but was then unable to credit your checking account, it would *roll-back* the transaction. In other words, the transaction would either be fully completed or any partial changes would be erased. The basic idea behind transactional memory is that critical sections in shared-memory programs should be treated as transactions. Either a thread successfully completes the critical section or any partial results are rolled back and the critical section is repeated.

Thread safety

In many, if not most, cases parallel programs can call functions developed for use in serial programs, and there won't be any problems. However, there are some notable exceptions. The most important exception for C programmers occurs in functions that make use of *static* local variables. Recall that ordinary C local variables—variables declared inside a function—are allocated from the system stack. Since each thread has its own stack, ordinary C local variables are private. However, recall that a static variable that's declared in a function persists from one call to the next. Thus, static variables are effectively shared among any threads that call the function, and this can have unexpected and unwanted consequences.

For example, the C string library function `strtok` splits an input string into substrings. When it's first called, it's passed a string, and on subsequent calls it returns successive substrings. This can be arranged through the use of a static `char*` variable that refers to the string that was passed on the first call. Now suppose two threads are splitting strings into substrings. Clearly, if, for example, thread 0 makes its first call to `strtok`, and then thread 1 makes its first call to `strtok` before thread 0 has completed splitting its string, then thread 0's string will be lost or overwritten, and, on subsequent calls it may get substrings of thread 1's strings.

A function such as `strtok` is not **thread safe**. This means that if it is used in a multithreaded program, there may be errors or unexpected results. When a block of code isn't thread safe, it's usually because different threads are accessing shared data. Thus, as we've seen, even though many serial functions can be used safely in multithreaded programs—that is, they're *thread safe*—programmers need to be wary of functions that were written exclusively for use in serial programs. We'll take a closer look at thread safety in Chapters 4 and 5.

2.4.4 Distributed-memory

In distributed-memory programs, the cores can directly access only their own, private memories. There are several APIs that are used. However, by far the most widely used is message-passing. So we'll devote most of our attention in this section to message-passing. Then we'll take a brief look at at a couple of other, less widely used, APIs.

Perhaps the first thing to note regarding distributed-memory APIs is that they can be used with shared-memory hardware. It's perfectly feasible for programmers to logically partition shared-memory into private address spaces for the various threads, and a library or compiler can implement the communication that's needed.

As we noted earlier, distributed-memory programs are usually executed by starting multiple processes rather than multiple threads. This is because typical "threads of execution" in a distributed-memory program may run on independent CPUs with independent operating systems, and there may be no software infrastructure for starting a single "distributed" process and having that process fork one or more threads on each node of the system.

Message-passing

A message-passing API provides (at a minimum) a send and a receive function. Processes typically identify each other by ranks in the range $0, 1, \ldots, p-1$, where p is the number of processes. So, for example, process 1 might send a message to process 0 with the following pseudo-code:

```
char message[100];
. . .
my_rank = Get_rank();
if (my_rank == 1) {
   sprintf(message, "Greetings from process 1");
   Send(message, MSG_CHAR, 100, 0);
} else if (my_rank == 0) {
   Receive(message, MSG_CHAR, 100, 1);
   printf("Process 0 > Received: %s\n", message);
}
```

Here the `Get_rank` function returns the calling process' rank. Then the processes branch depending on their ranks. Process 1 creates a message with `sprintf` from the standard C library and then sends it to process 0 with the call to `Send`. The arguments to the call are, in order, the message, the type of the elements in the message (`MSG_CHAR`), the number of elements in the message (100), and the rank of the destination process (0). On the other hand, process 0 calls `Receive` with the following arguments: the variable into which the message will be received (`message`), the type of the message elements, the number of elements available for storing the message, and the rank of the process sending the message. After completing the call to `Receive`, process 0 prints the message.

Several points are worth noting here. First note that the program segment is SPMD. The two processes are using the same executable, but carrying out different actions. In this case, what they do depends on their ranks. Second, note that the variable `message` refers to different blocks of memory on the different processes. Programmers often stress this by using variable names such as `my_message` or `local_message`. Finally, note that we're assuming that process 0 can write to `stdout`. This is usually the case: most implementations of message-passing APIs allow all processes access to `stdout` and `stderr`—even if the API doesn't explicitly provide for this. We'll talk a little more about I/O later on.

There are several possibilities for the exact behavior of the `Send` and `Receive` functions, and most message-passing APIs provide several different send and/or receive functions. The simplest behavior is for the call to `Send` to **block** until the call to `Receive` starts receiving the data. This means that the process calling `Send` won't return from the call until the matching call to `Receive` has started. Alternatively, the `Send` function may copy the contents of the message into storage that it owns, and then it will return as soon as the data is copied. The most common behavior for the `Receive` function is for the receiving process to block until the message is received. There are other possibilities for both `Send` and `Receive`, and we'll discuss some of them in Chapter 3.

Typical message-passing APIs also provide a wide variety of additional functions. For example, there may be functions for various "collective" communications, such as a **broadcast**, in which a single process transmits the same data to all the processes, or a **reduction**, in which results computed by the individual processes are combined into a single result—for example, values computed by the processes are added. There may also be special functions for managing processes and communicating complicated data structures. The most widely used API for message-passing is the **Message-Passing Interface** or MPI. We'll take a closer look at it in Chapter 3.

Message-passing is a very powerful and versatile API for developing parallel programs. Virtually all of the programs that are run on the most powerful computers in the world use message-passing. However, it is also very low level. That is, there is a huge amount of detail that the programmer needs to manage. For example, in order to parallelize a serial program, it is usually necessary to rewrite the vast majority of the program. The data structures in the program may have to either be replicated by each process or be explicitly distributed among the processes. Furthermore, the rewriting usually can't be done incrementally. For example, if a data structure is used in many parts of the program, distributing it for the parallel parts and collecting it for the serial (unparallelized) parts will probably be prohibitively expensive. Therefore, message-passing is sometimes called "the assembly language of parallel programming," and there have been many attempts to develop other distributed-memory APIs.

One-sided communication

In message-passing, one process, must call a send function and the send must be matched by another process' call to a receive function. Any communication requires the explicit participation of two processes. In **one-sided communication**, or **remote memory access**, a single process calls a function, which updates either local memory with a value from another process or remote memory with a value from the calling process. This can simplify communication, since it only requires the active participation of a single process. Furthermore, it can significantly reduce the cost of communication by eliminating the overhead associated with synchronizing two processes. It can also reduce overhead by eliminating the overhead of one of the function calls (send or receive).

It should be noted that some of these advantages may be hard to realize in practice. For example, if process 0 is copying a value into the memory of process 1, 0 must have some way of knowing when it's safe to copy, since it will overwrite some memory location. Process 1 must also have some way of knowing when the memory location has been updated. The first problem can be solved by synchronizing the two processes before the copy, and the second problem can be solved by another synchronization or by having a "flag" variable that process 0 sets after it has completed the copy. In the latter case, process 1 may need to **poll** the flag variable in order to determine that the new value is available. That is, it must repeatedly check the flag variable until it gets the value indicating 0 has completed its copy. Clearly, these problems can considerably increase the overhead associated with transmitting a value. A further difficulty is that since there is no explicit interaction between the two processes, remote memory operations can introduce errors that are very hard to track down.

Partitioned global address space languages

Since many programmers find shared-memory programming more appealing than message-passing or one-sided communication, a number of groups are developing parallel programming languages that allow the user to use some shared-memory techniques for programming distributed-memory hardware. This isn't quite as simple as it sounds. If we simply wrote a compiler that treated the collective memories in a distributed-memory system as a single large memory, our programs would have poor, or, at best, unpredictable performance, since each time a running process accessed memory, it might access local memory—that is, memory belonging to the core on which it was executing—or remote memory, memory belonging to another core. Accessing remote memory can take hundreds or even thousands of times longer than accessing local memory. As an example, consider the following pseudo-code for a shared-memory vector addition:

```
shared int n = . . . ;
shared double x[n], y[n];
private int i, my_first_element, my_last_element;
my_first_element = . . . ;
my_last_element = . . . ;
```

```
/* Initialize x and y */
. . .

for (i = my_first_element; i <= my_last_element; i++)
   x[i] += y[i];
```

We first declare two shared arrays. Then, on the basis of the process' rank, we determine which elements of the array "belong" to which process. After initializing the arrays, each process adds its assigned elements. If the assigned elements of `x` and `y` have been allocated so that the elements assigned to each process are in the memory attached to the core the process is running on, then this code should be very fast. However, if, for example, all of `x` is assigned to core 0 and all of `y` is assigned to core 1, then the performance is likely to be terrible, since each time the assignment `x[i] += y[i]` is executed, the process will need to refer to remote memory.

Partitioned global address space, or PGAS, languages provide some of the mechanisms of shared-memory programs. However, they provide the programmer with tools to avoid the problem we just discussed. Private variables are allocated in the local memory of the core on which the process is executing, and the distribution of the data in shared data structures is controlled by the programmer. So, for example, she knows which elements of a shared array are in which process' local memory.

There are a number of research projects currently working on the development of PGAS languages. See, for example, [7, 9, 45].

2.4.5 Programming hybrid systems

Before moving on, we should note that it is possible to program systems such as clusters of multicore processors using a combination of a shared-memory API on the nodes and a distributed-memory API for internode communication. However, this is usually only done for programs that require the highest possible levels of performance, since the complexity of this "hybrid" API makes program development extremely difficult. See, for example, [40]. Rather, such systems are usually programmed using a single, distributed-memory API for both inter- and intra-node communication.

2.5 INPUT AND OUTPUT

We've generally avoided the issue of input and output. There are a couple of reasons. First and foremost, parallel I/O, in which multiple cores access multiple disks or other devices, is a subject to which one could easily devote a book. See, for example, [35]. Second, the vast majority of the programs we'll develop do very little in the way of I/O. The amount of data they read and write is quite small and easily managed by the standard C I/O functions `printf`, `fprintf`, `scanf`, and `fscanf`. However, even the limited use we make of these functions can potentially cause some problems. Since these functions are part of standard C, which is a serial language, the standard says nothing about what happens when they're called by different processes. On the other hand, threads that are forked by a single process *do* share `stdin`, `stdout`, and `stderr`. However, (as we've seen), when multiple threads attempt to access one of these, the outcome is nondeterministic, and it's impossible to predict what will happen.

When we call `printf` from multiple processes, we, as developers, would like the output to appear on the console of a single system, the system on which we started the program. In fact, this is what the vast majority of systems do. However, there is no guarantee, and we need to be aware that it is possible for a system to do something else, for example, only one process has access to `stdout` or `stderr` or even *no* processes have access to `stdout` or `stderr`.

What *should* happen with calls to `scanf` when we're running multiple processes is a little less obvious. Should the input be divided among the processes? Or should only a single process be allowed to call `scanf`? The vast majority of systems allow at least one process to call `scanf`—usually process 0—while some allow more processes. Once again, there are some systems that don't allow any processes to call `scanf`.

When multiple processes *can* access `stdout`, `stderr`, or `stdin`, as you might guess, the distribution of the input and the sequence of the output are usually nondeterministic. For output, the data will probably appear in a different order each time the program is run, or, even worse, the output of one process may be broken up by the output of another process. For input, the data read by each process may be different on each run, even if the same input is used.

In order to partially address these issues, we'll be making these assumptions and following these rules when our parallel programs need to do I/O:

- In distributed-memory programs, only process 0 will access `stdin`. In shared-memory programs, only the master thread or thread 0 will access `stdin`.
- In both distributed-memory and shared-memory programs, all the processes/threads can access `stdout` and `stderr`.
- However, because of the nondeterministic order of output to `stdout`, in most cases only a single process/thread will be used for all output to `stdout`. The principal exception will be output for debugging a program. In this situation, we'll often have multiple processes/threads writing to `stdout`.
- Only a single process/thread will attempt to access any single file other than `stdin`, `stdout`, or `stderr`. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.
- Debug output should always include the rank or id of the process/thread that's generating the output.

2.6 PERFORMANCE

Of course our main purpose in writing parallel programs is usually increased performance. So what can we expect? And how can we evaluate our programs?

2.6.1 Speedup and efficiency

Usually the best we can hope to do is to equally divide the work among the cores, while at the same time introducing no additional work for the cores. If we succeed in doing this, and we run our program with p cores, one thread or process on each core, then our parallel program will run p times faster than the serial program. If we call the serial run-time T_{serial} and our parallel run-time T_{parallel}, then the best we can hope for is $T_{\text{parallel}} = T_{\text{serial}}/p$. When this happens, we say that our parallel program has **linear speedup**.

In practice, we're unlikely to get linear speedup because the use of multiple processes/threads almost invariably introduces some overhead. For example, shared-memory programs will almost always have critical sections, which will require that we use some mutual exclusion mechanism such as a mutex. The calls to the mutex functions are overhead that's not present in the serial program, and the use of the mutex forces the parallel program to serialize execution of the critical section. Distributed-memory programs will almost always need to transmit data across the network, which is usually much slower than local memory access. Serial programs, on the other hand, won't have these overheads. Thus, it will be very unusual for us to find that our parallel programs get linear speedup. Furthermore, it's likely that the overheads will increase as we increase the number of processes or threads, that is, more threads will probably mean more threads need to access a critical section. More processes will probably mean more data needs to be transmitted across the network.

So if we define the **speedup** of a parallel program to be

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},$$

then linear speedup has $S = p$, which is unusual. Furthermore, as p increases, we expect S to become a smaller and smaller fraction of the ideal, linear speedup p. Another way of saying this is that S/p will probably get smaller and smaller as p increases. Table 2.4 shows an example of the changes in S and S/p as p increases.[1]

This value, S/p, is sometimes called the **efficiency** of the parallel program. If we substitute the formula for S, we see that the efficiency is

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

[1]These data are taken from Chapter 3. See Tables 3.6 and 3.7.

Table 2.4 Speedups and Efficiencies of a Parallel Program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

Table 2.5 Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

It's clear that T_{parallel}, S, and E depend on p, the number of processes or threads. We also need to keep in mind that T_{parallel}, S, E, and T_{serial} all depend on the problem size. For example, if we halve and double the problem size of the program whose speedups are shown in Table 2.4, we get the speedups and efficiencies shown in Table 2.5. The speedups are plotted in Figure 2.18, and the efficiencies are plotted in Figure 2.19.

We see that in this example, when we increase the problem size, the speedups and the efficiencies increase, while they decrease when we decrease the problem size. This behavior is quite common. Many parallel programs are developed by dividing the work of the serial program among the processes/threads and adding in the necessary "parallel overhead" such as mutual exclusion or communication. Therefore, if T_{overhead} denotes this parallel overhead, it's often the case that

$$T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}.$$

Furthermore, as the problem size is increased, T_{overhead} often grows more slowly than T_{serial}. When this is the case the speedup and the efficiency will increase. See Exercise 2.16. This is what your intuition should tell you: there's more work for the processes/threads to do, so the relative amount of time spent coordinating the work of the processes/threads should be less.

A final issue to consider is what values of T_{serial} should be used when reporting speedups and efficiencies. Some authors say that T_{serial} should be the run-time of the fastest program on the fastest processor available. In practice, most authors use a serial program on which the parallel program was based and run it on a single

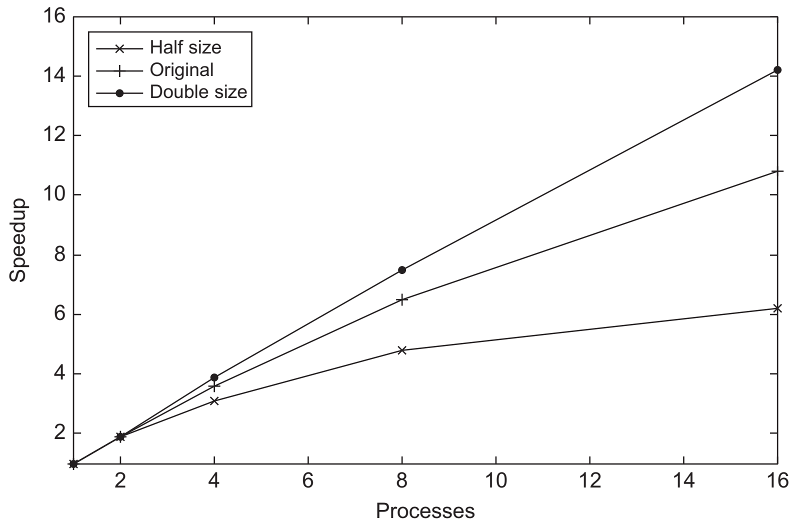

FIGURE 2.18

Speedups of parallel program on different problem sizes

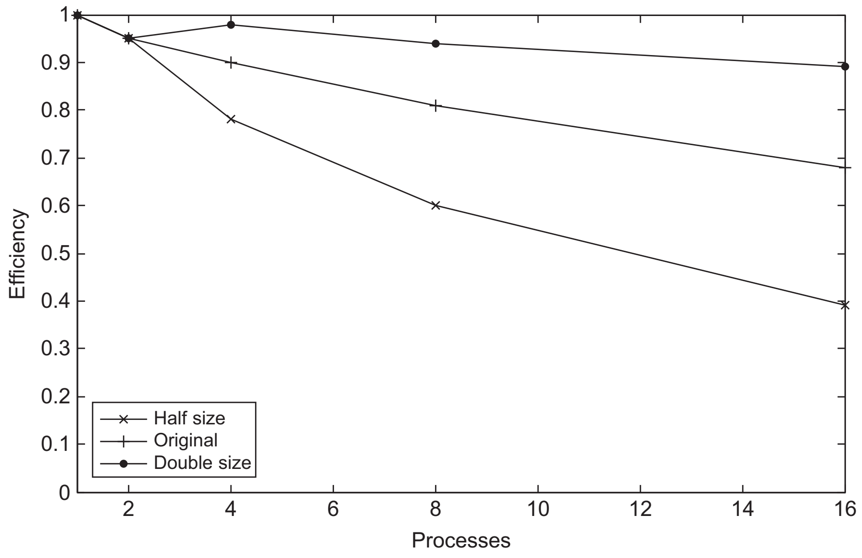

FIGURE 2.19

Efficiencies of parallel program on different problem sizes processor of the parallel system. So if we were studying the performance of a parallel shell sort program, authors in the first group might use a serial radix sort or quicksort on a single core of the fastest system available, while authors in the second group would use a serial shell sort on a single processor of the parallel system. We'll generally use the second approach.

2.6.2 Amdahl's law

Back in the 1960s, Gene Amdahl made an observation [2] that's become known as **Amdahl's law.** It says, roughly, that unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited—regardless of the number of cores available. Suppose, for example, that we're able to parallelize 90% of a serial program. Further suppose that the parallelization is "perfect," that is, regardless of the number of cores p we use, the speedup of this part of the program will be p. If the serial run-time is $T_{\text{serial}}= 20$ seconds, then the run-time of the parallelized part will be $0.9 \times T_{\text{serial}}/p = 18/p$ and the run-time of the "unparallelized" part will be $0.1 \times T_{\text{serial}} = 2$. The overall parallel run-time will be

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18/p + 2,$$

and the speedup will be

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p+2}.$$

Now as p gets larger and larger, $0.9 \times T_{\text{serial}}/p = 18/p$ gets closer and closer to 0, so the total parallel run-time can't be smaller than $0.1 \times T_{\text{serial}} = 2$. That is, the denominator in S can't be smaller than $0.1 \times T_{\text{serial}} = 2$. The fraction S must therefore be smaller than

$$S \leq \frac{T_{\text{serial}}}{0.1 \times T_{\text{serial}}} = \frac{20}{2} = 10.$$

That is, $S \leq 10$. This is saying that even though we've done a perfect job in parallelizing 90% of the program, and even if we have, say, 1000 cores, we'll never get a speedup better than 10.

More generally, if a fraction r of our serial program remains unparallelized, then Amdahl's law says we can't get a speedup better than $1/r$. In our example, $r = 1 - 0.9 = 1/10$, so we couldn't get a speedup better than 10. Therefore, if a fraction r of our serial program is "inherently serial," that is, cannot possibly be parallelized, then we can't possibly get a speedup better than $1/r$. Thus, even if r is quite small—say 1/100—and we have a system with thousands of cores, we can't possibly get a speedup better than 100.

This is pretty daunting. Should we give up and go home? Well, no. There are several reasons not to be too worried by Amdahl's law. First, it doesn't take into consideration the problem size. For many problems, as we increase the problem size, the "inherently serial" fraction of the program decreases in size; a more mathematical version of this statement is known as **Gustafson's law** [25]. Second, there are thousands of programs used by scientists and engineers that routinely obtain huge speedups on large distributed-memory systems. Finally, is a small speedup so awful? In many cases, obtaining a speedup of 5 or 10 is more than adequate, especially if the effort involved in developing the parallel program wasn't very large.

2.6.3 Scalability

The word "scalable" has a wide variety of informal uses. Indeed, we've used it several times already. Roughly speaking, a technology is scalable if it can handle ever-increasing problem sizes. However, in discussions of parallel program performance, scalability has a somewhat more formal definition. Suppose we run a parallel program with a fixed number of processes/threads and a fixed input size, and we obtain an efficiency E. Suppose we now increase the number of processes/threads that are used by the program. If we can find a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is **scalable**.

As an example, suppose that $T_{\text{serial}} = n$, where the units of T_{serial} are in microseconds, and n is also the problem size. Also suppose that $T_{\text{parallel}} = n/p + 1$. Then

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}.$$

To see if the program is scalable, we increase the number of processes/threads by a factor of k, and we want to find the factor x that we need to increase the problem size by so that E is unchanged. The number of processes/threads will be kp and the problem size will be xn, and we want to solve the following equation for x:

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

Well, if $x = k$, there will be a common factor of k in the denominator $xn + kp = kn + kp = k(n+p)$, and we can reduce the fraction to get

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}.$$

In other words, if we increase the problem size at the same rate that we increase the number of processes/threads, then the efficiency will be unchanged, and our program is scalable.

There are a couple of cases that have special names. If when we increase the number of processes/threads, we can keep the efficiency fixed *without* increasing the problem size, the program is said to be *strongly scalable*. If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, then the program is said to be *weakly scalable*. The program in our example would be weakly scalable.

2.6.4 Taking timings

You may have been wondering how we find T_{serial} and T_{parallel}. There are a *lot* of different approaches, and with parallel programs the details may depend on the API. However, there are a few general observations we can make that may make things a little easier.

The first thing to note is that there are at least two different reasons for taking timings. During program development we may take timings in order to determine if the program is behaving as we intend. For example, in a distributed-memory program we might be interested in finding out how much time the processes are spending waiting for messages, because if this value is large, there is almost certainly something wrong either with our design or our implementation. On the other hand, once we've completed development of the program, we're often interested in determining how good its performance is. Perhaps surprisingly, the way we take these two timings is usually different. For the first timing, we usually need very detailed information: How much time did the program spend in this part of the program? How much time did it spend in that part? For the second, we usually report a single value. Right now we'll talk about the second type of timing. See Exercise 2.22 for a brief discussion of some issues in taking the first type of timing.

Second, we're usually *not* interested in the time that elapses between the program's start and the program's finish. We're usually interested only in some part of the program. For example, if we write a program that implements bubble sort, we're probably only interested in the time it takes to sort the keys, not the time it takes to read them in and print them out. We probably can't use something like the Unix shell command `time`, which reports the time taken to run a program from start to finish.

Third, we're usually *not* interested in "CPU time." This is the time reported by the standard C function `clock`. It's the total time the program spends in code executed as part of the program. It would include the time for code we've written; it would include the time we spend in library functions such as `pow` or `sin`; and it would include the time the operating system spends in functions we call, such as `printf` and `scanf`. It would not include time the program was idle, and this could be a problem. For example, in a distributed-memory program, a process that calls a receive function may have to wait for the sending process to execute the matching send, and the operating system might put the receiving process to sleep while it waits. This idle time wouldn't be counted as CPU time, since no function that's been called by the process is active. However, it should count in our evaluation of the overall run-time, since it may be a real cost in our program. If each time the program is run, the process has to wait, ignoring the time it spends waiting would give a misleading picture of the actual run-time of the program.

Thus, when you see an article reporting the run-time of a parallel program, the reported time is usually "wall clock" time. That is, the authors of the article report the time that has elapsed between the start and finish of execution of the code that the user is interested in. If the user could see the execution of the program, she would hit the start button on her stopwatch when it begins execution and hit the stop button when it stops execution. Of course, she can't see her code executing, but she can modify the source code so that it looks something like this:

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

The function `Get_current_time()` is a hypothetical function that's supposed to return the number of seconds that have elapsed since some fixed time in the past. It's just a placeholder. The actual function that is used will depend on the API. For example, MPI has a function `MPI_Wtime` that could be used here, and the OpenMP API for shared-memory programming has a function `omp_get_wtime`. Both functions return wall clock time instead of CPU time.

There may be an issue with the **resolution** of the timer function. The resolution is the unit of measurement on the timer. It's the duration of the shortest event that can have a nonzero time. Some timer functions have resolutions in milliseconds (10^{-3} seconds), and when instructions can take times that are less than a nanosecond (10^{-9} seconds), a program may have to execute millions of instructions before the timer reports a nonzero time. Many APIs provide a function that reports the resolution of the timer. Other APIs specify that a timer must have a given resolution. In either case we, as the programmers, need to check these values.

When we're timing parallel programs, we need to be a little more careful about how the timings are taken. In our example, the code that we want to time is probably being executed by multiple processes or threads and our original timing will result in the output of p elapsed times.

```
private double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

However, what we're usually interested in is a single time: the time that has elapsed from when the first process/thread began execution of the code to the time the last process/thread finished execution of the code. We often can't obtain this exactly, since there may not be any correspondence between the clock on one node and the clock on another node. We usually settle for a compromise that looks something like this:

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
```

```
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish − my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n", global_elapsed);
```

Here, we first execute a **barrier** function that approximately synchronizes all of the processes/threads. We would like for all the processes/threads to return from the call simultaneously, but such a function usually can only guarantee that all the processes/threads have started the call when the first process/thread returns. We then execute the code as before and each process/thread finds the time it took. Then all the processes/threads call a global maximum function, which returns the largest of the elapsed times, and process/thread 0 prints it out.

We also need to be aware of the *variability* in timings. When we run a program several times, it's extremely likely that the elapsed time will be different for each run. This will be true even if each time we run the program we use the same input and the same systems. It might seem that the best way to deal with this would be to report either a mean or a median run-time. However, it's unlikely that some outside event could actually make our program run faster than its best possible run-time. So instead of reporting the mean or median time, we usually report the *minimum* time.

Running more than one thread per core can cause dramatic increases in the variability of timings. More importantly, if we run more than one thread per core, the system will have to take extra time to schedule and deschedule cores, and this will add to the overall run-time. Therefore, we rarely run more than one thread per core.

Finally, as a practical matter, since our programs won't be designed for high-performance I/O, we'll usually not include I/O in our reported run-times.

2.7 PARALLEL PROGRAM DESIGN

So we've got a serial program. How do we parallelize it? We know that in general we need to divide the work among the processes/threads so that each process gets roughly the same amount of work and communication is minimized. In most cases, we also need to arrange for the processes/threads to synchronize and communicate.

Unfortunately, there isn't some mechanical process we can follow; if there were, we could write a program that would convert any serial program into a parallel program, but, as we noted in Chapter 1, in spite of a tremendous amount of work and some progress, this seems to be a problem that has no universal solution.

However, Ian Foster provides an outline of steps in his online book *Designing and Building Parallel Programs* [19]:

1. *Partitioning*. Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. *Communication*. Determine what communication needs to be carried out among the tasks identified in the previous step.
3. *Agglomeration or aggregation*. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. *Mapping*. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

This is sometimes called **Foster's methodology**.

2.7.1 An example

Let's look at a small example. Suppose we have a program that generates large quantities of floating point data that it stores in an array. In order to get some feel for the distribution of the data, we can make a histogram of the data. Recall that to make a histogram, we simply divide the range of the data up into equal sized subintervals, or *bins*, determine the number of measurements in each bin, and plot a bar graph showing the relative sizes of the bins. As a very small example, suppose our data are

$$1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9.$$

Then the data lie in the range 0–5, and if we choose to have five bins, the histogram might look something like Figure 2.20.

A serial program

It's pretty straightforward to write a serial program that generates a histogram. We need to decide what the bins are, determine the number of measurements in each bin, and print the bars of the histogram. Since we're not focusing on I/O, we'll limit ourselves to just the first two steps, so the input will be

1. the number of measurements, `data_count`;
2. an array of `data_count` floats, `data`;
3. the minimum value for the bin containing the smallest values, `min_meas`;
4. the maximum value for the bin containing the largest values, `max_meas`;
5. the number of bins, `bin_count`;

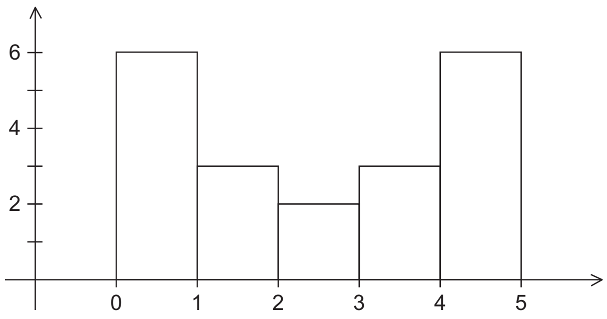

FIGURE 2.20

A histogram

The output will be an array containing the number of elements of data that lie in each bin. To make things precise, we'll make use of the following data structures:

– `bin_maxes`. An array of `bin_count` floats
– `bin_counts`. An array of `bin_count` ints

The array `bin_maxes` will store the upper bound for each bin, and `bin_counts` will store the number of data elements in each bin. To be explicit, we can define

```
        bin_width = (max_meas − min_meas)/bin_count
```

Then `bin_maxes` will be initialized by

```
      for (b = 0; b < bin_count; b++)
         bin_maxes[b] = min_meas + bin_width*(b+1);
```

We'll adopt the convention that bin b will be all the measurements in the range

```
   bin_maxes[b−1] <= measurement < bin_maxes[b]
```

Of course, this doesn't make sense if $b = 0$, and in this case we'll use the rule that bin 0 will be the measurements in the range

```
   min_meas <= measurement < bin_maxes[0]
```

This means we always need to treat bin 0 as a special case, but this isn't too onerous.

Once we've initialized `bin_maxes` and assigned 0 to all the elements of `bin_counts`, we can get the counts by using the following pseudo-code:

```
   for (i = 0; i < data_count; i++) {
      bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);
      bin_counts[bin]++;
   }
```

The `Find_bin` function returns the bin that `data[i]` belongs to. This could be a simple linear search function: search through `bin_maxes` until you find a bin b that satisfies

```
      bin_maxes[b−1] <= data[i] < bin_maxes[b]
```

(Here we're thinking of `bin_maxes[−1]` as `min_meas`.) This will be fine if there aren't very many bins, but if there are a lot of bins, binary search will be much better.

Parallelizing the serial program

If we assume that `data_count` is much larger than `bin_count`, then even if we use binary search in the `Find_bin` function, the vast majority of the work in this code will be in the loop that determines the values in `bin_counts`. The focus of our parallelization should therefore be on this loop, and we'll apply Foster's methodology to it. The first thing to note is that the outcomes of the steps in Foster's methodology are by no means uniquely determined, so you shouldn't be surprised if at any stage you come up with something different.

For the first step we might identify two types of tasks: finding the bin to which an element of `data` belongs and incrementing the appropriate entry in `bin_counts`.

For the second step, there must be a communication between the computation of the appropriate bin and incrementing an element of `bin_counts`. If we represent our tasks with ovals and communications with arrows, we'll get a diagram that looks something like that shown in Figure 2.21. Here, the task labelled with "`data[i]`" is determining which bin the value `data[i]` belongs to, and the task labelled with "`bin_counts[b]++`" is incrementing `bin_counts[b]`.

For any fixed element of `data`, the tasks "find the bin `b` for element of `data`" and "increment `bin_counts[b]`" can be aggregated, since the second can only happen once the first has been completed.

However, when we proceed to the final or mapping step, we see that if two processes or threads are assigned elements of `data` that belong to the same bin `b`, they'll both result in execution of the statement `bin_counts[b]++`. If `bin_counts[b]` is shared (e.g., the array `bin_counts` is stored in shared-memory), then this will result in a race condition. If `bin_counts` has been partitioned among the processes/threads, then updates to its elements will require communication. An alternative is to store multiple "local" copies of `bin_counts` and add the values in the local copies after all the calls to `Find_bin`.

If the number of bins, `bin_count`, isn't absolutely gigantic, there shouldn't be a problem with this. So let's pursue this alternative, since it is suitable for use on both shared- and distributed-memory systems.

In this setting, we need to update our diagram so that the second collection of tasks increments `loc_bin_cts[b]`. We also need to add a third collection of tasks, adding the various `loc_bin_cts[b]` to get `bin_counts[b]`. See Figure 2.22. Now we

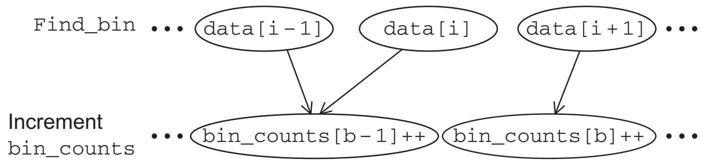

FIGURE 2.21

The first two stages of Foster's methodology

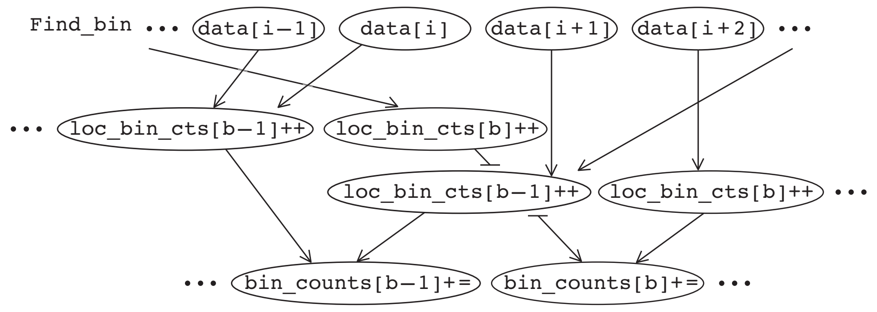

FIGURE 2.22

Alternative definition of tasks and communication

see that if we create an array `loc_bin_cts` for each process/thread, then we can map the tasks in the first two groups as follows:

1. Elements of `data` are assigned to the processes/threads so that each process/thread gets roughly the same number of elements.
2. Each process/thread is responsible for updating its `loc_bin_cts` array on the basis of its assigned elements.

To finish up, we need to add the elements `loc_bin_cts[b]` into `bin_counts[b]`. If both the number of processes/threads is small and the number of bins is small, all of the additions can be assigned to a single process/thread. If the number of bins is much larger than the number of processes/threads, we can divide the bins among the processes/threads in much the same way that we divided the elements of `data`. If the number of processes/threads is large, we can use a tree-structured global sum similar to the one we discussed in Chapter 1. The only difference is that now the sending process/threads are sending an array, and the receiving process/threads are receiving and adding an array. Figure 2.23 shows an example with eight processes/threads. Each

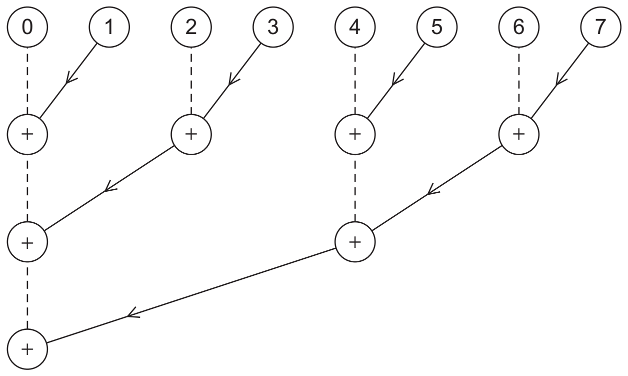

FIGURE 2.23

Adding the local arrays

circle in the top row corresponds to a process/thread. Between the first and the second rows, the odd-numbered processes/threads make their `loc_bin_cts` available to the even-numbered processes/threads. Then in the second row, the even-numbered processes/threads add the new counts to their existing counts. Between the second and third rows the process is repeated with the processes/threads whose ranks aren't divisible by four sending to those whose are. This process is repeated until process/thread 0 has computed `bin_counts`.

2.8 WRITING AND RUNNING PARALLEL PROGRAMS

In the past, virtually all parallel program development was done using a text editor such as `vi` or `Emacs`, and the program was either compiled and run from the command line or from within the editor. Debuggers were also typically started from the command line. Now there are also integrated development environments (IDEs) available from Microsoft, the Eclipse project, and others; see [16, 38].

In smaller shared-memory systems, there is a single running copy of the operating system, which ordinarily schedules the threads on the available cores. On these systems, shared-memory programs can usually be started using either an IDE or the command line. Once started, the program will typically use the console and the keyboard for input from `stdin` and output to `stdout` and `stderr`. On larger systems, there may be a batch scheduler, that is, a user requests a certain number of cores, and specifies the path to the executable and where input and output should go (typically to files in secondary storage).

In typical distributed-memory and hybrid systems, there is a host computer that is responsible for allocating nodes among the users. Some systems are purely *batch* systems, which are similar to shared-memory batch systems. Others allow users to check out nodes and run jobs interactively. Since job startup often involves communicating with remote systems, the actual startup is usually done with a script. For example, MPI programs are usually started with a script called `mpirun` or `mpiexec`.

As usual, RTFD, which is sometimes translated as "read the fine documentation."

2.9 ASSUMPTIONS

As we noted earlier, we'll be focusing on homogeneous MIMD systems—systems in which all of the nodes have the same architecture—and our programs will be SPMD. Thus, we'll write a single program that can use branching to have multiple different behaviors. We'll assume the cores are identical but that they operate asynchronously. We'll also assume that we always run at most one process or thread of our program on a single core, and we'll often use static processes or threads. In other words, we'll often start all of our processes or threads at more or less the same time, and when they're done executing, we'll terminate them at more or less the same time.

Some application programming interfaces or APIs for parallel systems define new programming languages. However, most extend existing languages, either through a library of functions—for example, functions to pass messages—or extensions to the compiler for the serial language. This latter approach will be the focus of this text. We'll be using parallel extensions to the C language.

When we want to be explicit about compiling and running programs, we'll use the command line of a Unix shell, the `gcc` compiler or some extension of it (e.g., `mpicc`), and we'll start programs from the command line. For example, if we wanted to show compilation and execution of the "hello, world" program from Kernighan and Ritchie [29], we might show something like this:

```
$ gcc −g −Wall −o hello hello.c
$ ./hello
hello, world
```

The $-sign is the prompt from the shell. We will usually use the following options for the compiler:

- `−g`. Create information that allows us to use a debugger
- `−Wall`. Issue lots of warnings
- `−o <outfile>`. Put the executable in the file named `outfile`
- When we're timing programs, we usually tell the compiler to optimize the code by using the `−O2` option.

In most systems, user directories or folders are not, by default, in the user's execution path, so we'll usually start jobs by giving the path to the executable by adding `./` to its name.

2.10 SUMMARY

There's a *lot* of material in this chapter, and a complete summary would run on for many pages, so we'll be very terse.

2.10.1 Serial systems

We started with a discussion of conventional serial hardware and software. The standard model of computer hardware has been the **von Neumann architecture**, which consists of a **central processing unit** that carries out the computations, and **main memory** that stores data and instructions. The separation of CPU and memory is often called the **von Neumann bottleneck** since it limits the rate at which instructions can be executed.

Perhaps the most important software on a computer is the **operating system**. It manages the computer's resources. Most modern operating systems are **multitasking**. Even if the hardware doesn't have multiple processors or cores, by rapidly switching among executing programs, the OS creates the illusion that multiple jobs are running simultaneously. A running program is called a **process**. Since a running process is more or less autonomous, it has a lot of associated data and information. A **thread** is started by a process. A thread doesn't need as much associated data and information, and threads can be stopped and started much faster than processes.

In computer science, **caches** are memory locations that can be accessed faster than other memory locations. CPU caches are memory locations that are intermediate between the CPU registers and main memory. Their purpose is to reduce the delays associated with accessing main memory. Data are stored in cache using the principle of locality, that is, items that are close to recently accessed items are more likely to be accessed in the near future. Thus, contiguous **blocks** or **lines** of data and instructions are transferred between main memory and caches. When an instruction or data item is accessed and it's already in cache, it's called a cache **hit**. If the item isn't in cache, it's called a cache **miss**. Caches are directly managed by the computer hardware, so programmers can only indirectly control caching.

Main memory can also function as a cache for secondary storage. This is managed by the hardware and the operating system through **virtual memory**. Rather than storing all of a program's instructions and data in main memory, only the active parts are stored in main memory, and the remaining parts are stored in secondary storage called **swap space**. Like CPU caches, virtual memory operates on blocks of contiguous data and instructions, which in this setting are called **pages**. Note that instead of addressing the memory used by a program with physical addresses, virtual memory uses **virtual addresses**, which are independent of actual physical addresses. The correspondence between physical addresses and virtual addresses is stored in main memory in a **page table**. The combination of virtual addresses and a page table provides the system with the flexibility to store a program's data and instructions anywhere in memory. Thus, it won't matter if two different programs use the same virtual addresses. With the page table stored in main memory, it could happen that every time a program needed to access a main memory location it would need two memory accesses: one to get the appropriate page table entry so it could find the location in main memory, and one to actually access the desired memory. In order to avoid this problem, CPUs have a special page table cache called the **translation lookaside buffer**, which stores the most recently used page table entries.

Instruction-level parallelism (ILP) allows a single processor to execute multiple instructions simultaneously. There are two main types of ILP: **pipelining** and **multiple issue**. With pipelining, some of the functional units of a processor are ordered in sequence, with the output of one being the input to the next. Thus, while one piece of data is being processed by, say, the second functional unit, another piece of data can be processed by the first. With multiple issue, the functional units are replicated, and the processor attempts to simultaneously execute different instructions in a single program.

Rather than attempting to simultaneously execute individual instructions, **hardware multithreading** attempts to simultaneously execute different threads. There are several different approaches to implementing hardware multithreading. However, all of them attempt to keep the processor as busy as possible by rapidly switching between threads. This is especially important when a thread **stalls** and has to wait (e.g., for a memory access to complete) before it can execute an instruction. In **simultaneous multithreading**, the different threads can simultaneously use the multiple functional units in a multiple issue processor. Since threads consist of multiple instructions, we sometimes say that **thread-level parallelism**, or TLP, is **coarser-grained** than ILP.

2.10.2 Parallel hardware

ILP and TLP provide parallelism at a very low level; they're typically controlled by the processor and the operating system, and their use isn't directly controlled by the programmer. For our purposes, hardware is parallel if the parallelism is visible to the programmer, and she can modify her source code to exploit it.

Parallel hardware is often classified using **Flynn's taxonomy**, which distinguishes between the number of instruction streams and the number of data streams a system can handle. A von Neumann system has a single instruction stream and a single data stream so it is classified as a **single instruction, single data**, or **SISD**, system.

A **single instruction, multiple data**, or **SIMD**, system executes a single instruction at a time, but the instruction can operate on multiple data items. These systems often execute their instructions in lockstep: the first instruction is applied to all of the data elements simultaneously, then the second is applied, and so on. This type of parallel system is usually employed in **data parallel** programs, programs in which the data are divided among the processors and each data item is subjected to more or less the same sequence of instructions. **Vector processors** and **graphics processing units** are often classified as SIMD systems, although current generation GPUs also have characteristics of multiple instruction, multiple data stream systems.

Branching in SIMD systems is handled by idling those processors that might operate on a data item to which the instruction doesn't apply. This behavior makes SIMD systems poorly suited for **task-parallelism**, in which each processor executes a different task, or even data-parallelism, with many conditional branches.

As the name suggests, **multiple instruction, multiple data**, or **MIMD**, systems execute multiple independent instruction streams, each of which can have its own data stream. In practice, MIMD systems are collections of autonomous processors that can execute at their own pace. The principal distinction between different MIMD systems is whether they are **shared-memory** or **distributed-memory** systems. In shared-memory systems, each processor or core can directly access every memory location, while in distributed-memory systems, each processor has its own private memory. Most of the larger MIMD systems are **hybrid** systems in which a number of relatively small shared-memory systems are connected by an interconnection network. In such systems, the individual shared-memory systems are sometimes called **nodes**. Some MIMD systems are **heterogeneous** systems, in which the processors have different capabilities. For example, a system with a conventional CPU and a GPU is a heterogeneous system. A system in which all the processors have the same architecture is **homogeneous**.

There are a number of different interconnects for joining processors to memory in shared-memory systems and for interconnecting the processors in distributed-memory or hybrid systems. The most commonly used interconnects for shared-memory are **buses** and **crossbars**. Distributed-memory systems sometimes use **direct** interconnects such as **toroidal meshes** and **hypercubes**, and they sometimes use **indirect** interconnects such as crossbars and **multistage** networks. Networks are often evaluated by examining the **bisection width** or the **bisection bandwidth** of the network. These give measures of how much simultaneous communication the network can support. For individual communications between nodes, authors often discuss the **latency** and **bandwidth** of the interconnect.

A potential problem with shared-memory systems is **cache coherence**. The same variable can be stored in the caches of two different cores, and if one core updates the value of the variable, the other core may be unaware of the change. There are two main methods for insuring cache coherence: **snooping** and the use of **directories**. Snooping relies on the capability of the interconnect to broadcast information from each cache controller to every other cache controller. Directories are special distributed data structures, which store information on each cache line. Cache coherence introduces another problem for shared-memory programming: **false sharing**. When one core updates a variable in one cache line, and another core wants to access *another* variable in the same cache line, it will have to access main memory, since the unit of cache coherence is the cache line. That is, the second core only "knows" that the line it wants to access has been updated. It doesn't know that the variable it wants to access hasn't been changed.

2.10.3 Parallel software

In this text we'll focus on developing software for homogeneous MIMD systems. Most programs for such systems consist of a single program that obtains parallelism by branching. Such programs are often called **single program, multiple data** or **SPMD** programs. In shared-memory programs we'll call the instances of running tasks threads; in distributed-memory programs we'll call them processes.

Unless our problem is **embarrassingly parallel**, the development of a parallel program needs at a minimum to address the issues of **load balance**, **communication**, and **synchronization** among the processes or threads.

In shared-memory programs, the individual threads can have **private** and **shared**-memory. Communication is usually done through **shared variables**. Any time the processors execute asynchronously, there is the potential for **nondeterminism**, that is, for a given input the behavior of the program can change from one run to the next. This can be a serious problem, especially in shared-memory programs. If the nondeterminism results from two threads' attempts to access the same resource, and it can result in an error, the program is said to have a **race condition**. The most common place for a race condition is a **critical section**, a block of code that can only be executed by one thread at a time. In most shared-memory APIs, **mutual exclusion** in a critical section can be enforced with an object called a **mutual exclusion lock** or **mutex**. Critical sections should be made as small as possible, since a mutex will allow only one thread at a time to execute the code in the critical section, effectively making the code serial.

A second potential problem with shared-memory programs is **thread safety**. A block of code that functions correctly when it is run by multiple threads is said to be **thread safe**. Functions that were written for use in serial programs can make unwitting use of shared data—for example, static variables—and this use in a multithreaded program can cause errors. Such functions are not thread safe.

The most common API for programming distributed-memory systems is **message-passing**. In message-passing, there are (at least) two distinct functions: a **send** function and a **receive** function. When processes need to communicate, one calls the send and the other calls the receive. There are a variety of possible behaviors for these functions. For example, the send can **block** or wait until the matching receive has started, or the message-passing software can can copy the data for the message into its own storage, and the sending process can return before the matching receive has started. The most common behavior for receives is to block until the message has been received. The most commonly used message-passing system is called the **Message-Passing Interface** or MPI. It provides a great deal of functionality beyond simple sends and receives.

Distributed-memory systems can also be programmed using **one-sided communications**, which provide functions for accessing memory belonging to another process, and **partitioned global address space** languages, which provide some shared-memory functionality in distributed-memory systems.

2.10.4 Input and output

In general parallel systems, multiple cores can access multiple secondary storage devices. We won't attempt to write programs that make use of this functionality. Rather, we'll write programs in which one process or thread can access `stdin`, and all processes can access `stdout` and `stderr`. However, because of nondeterminism, except for debug output we'll usually have a single process or thread accessing `stdout`.

2.10.5 Performance

If we run a parallel program with p processes or threads and no more than one process/thread per core, then our ideal would be for our parallel program to run p times faster than the serial program. This is called **linear speedup**, but in practice it is rarely achieved. If we denote the run-time of the serial program by T_{serial} and the parallel program's run-time by T_{parallel}, then the **speedup** S and **efficiency** E of the parallel program are given by the formulas

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}, \text{ and } E = \frac{T_{\text{serial}}}{pT_{\text{parallel}}},$$

respectively. So linear speedup has $S = p$ and $E = 1$. In practice, we will almost always have $S < p$ and $E < 1$. If we fix the problem size, E usually decreases as we increase p, while if we fix the number of processes/threads, then S and E often increase as we increase the problem size.

Amdahl's law provides an upper bound on the speedup that can be obtained by a parallel program: if a fraction r of the original, serial program isn't parallelized, then we can't possibly get a speedup better than $1/r$, regardless of how many processes/threads we use. In practice many parallel programs obtain excellent speedups. One possible reason for this apparent contradiction is that Amdahl's law doesn't take into consideration the fact that the unparallelized part often decreases in size relative to the parallelized part as the problem size increases.

Scalability is a term that has many interpretations. In general, a technology is scalable if it can handle ever-increasing problem sizes. Formally, a parallel program is scalable if there is a rate at which the problem size can be increased so that as the number of processes/threads is increased, the efficiency remains constant. A program is **strongly** scalable, if the problem size can remain fixed, and it's **weakly** scalable if the problem size needs to be increased at the same rate as the number of processes/threads.

In order to determine T_{serial} and T_{parallel}, we usually need to include calls to a timer function in our source code. We want these timer functions to give **wall clock** time, not CPU time, since the program may be "active"—for example, waiting for a message—even when the CPU is idle. To take parallel times, we usually want to synchronize the processes/threads before starting the timer, and, after stopping the timer, find the maximum elapsed time among all the processes/threads. Because of system variability, we usually need to run a program several times with a given data set, and we usually take the minimum time from the multiple runs. To reduce variability and improve overall run-times, we usually run no more than one thread per core.

2.10.6 Parallel program design

Foster's methodology provides a sequence of steps that can be used to design parallel programs. The steps are **partitioning** the problem to identify tasks, identifying **communication** among the tasks, **agglomeration** or **aggregation** to group tasks, and **mapping** to assign aggregate tasks to processes/threads.

2.10.7 Assumptions

We'll be focusing on the development of parallel programs for both shared- and distributed-memory MIMD systems. We'll write SPMD programs that usually use **static** processes or threads—processes/threads that are created when the program begins execution, and are not shut down until the program terminates. We'll also assume that we run at most one process or thread on each core of the system.

2.11 EXERCISES

2.1. When we were discussing floating point addition, we made the simplifying assumption that each of the functional units took the same amount of time. Suppose that fetch and store each take 2 nanoseconds and the remaining operations each take 1 nanosecond.

- **a.** How long does a floating point addition take with these assumptions?
- **b.** How long will an unpipelined addition of 1000 pairs of floats take with these assumptions?
- **c.** How long will a pipelined addition of 1000 pairs of floats take with these assumptions?
- **d.** The time required for fetch and store may vary considerably if the operands/results are stored in different levels of the memory hierarchy. Suppose that a fetch from a level 1 cache takes two nanoseconds, while a fetch from a level 2 cache takes five nanoseconds, and a fetch from main memory takes fifty nanoseconds. What happens to the pipeline when there is a level 1 cache miss on a fetch of one of the operands? What happens when there is a level 2 miss?

2.2. Explain how a queue, implemented in hardware in the CPU, could be used to improve the performance of a write-through cache.

2.3. Recall the example involving cache reads of a two-dimensional array (page 22). How does a larger matrix and a larger cache affect the performance of the two pairs of nested loops? What happens if `MAX = 8` and the cache can store four lines? How many misses occur in the reads of `A` in the first pair of nested loops? How many misses occur in the second pair?

2.4. In Table 2.2, virtual addresses consist of a byte offset of 12 bits and a virtual page number of 20 bits. How many pages can a program have if it's run on a system with this page size and this virtual address size?

2.5. Does the addition of cache and virtual memory to a von Neumann system change its designation as an SISD system? What about the addition of pipelining? Multiple issue? Hardware multithreading?

2.6. Suppose that a vector processor has a memory system in which it takes 10 cycles to load a single 64-bit word from memory. How many memory banks are needed so that a stream of loads can, on average, require only one cycle per load?

2.7. Discuss the differences in how a GPU and a vector processor might execute the following code:

```
sum = 0.0;
for (i = 0; i < n; i++) {
   y[i] += a*x[i];
   sum += z[i]*z[i];
}
```

2.8. Explain why the performance of a hardware multithreaded processing core might degrade if it had large caches and it ran many threads.

2.9. In our discussion of parallel hardware, we used Flynn's taxonomy to identify three types of parallel systems: SISD, SIMD, and MIMD. None of our systems were identified as multiple instruction, single data, or MISD. How would an MISD system work? Give an example.

2.10. Suppose a program must execute 10^{12} instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in 10^6 seconds (about 11.6 days). So, on average, the single processor system executes 10^6 or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute $10^{12}/p$ instructions and each processor must send $10^9(p-1)$ messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all of its messages, and there won't be any delays due to things such as waiting for messages.

a. Suppose it takes 10^{-9} seconds to send a message. How long will it take the program to run with 1000 processors, if each processor is as fast as the single processor on which the serial program was run?

b. Suppose it takes 10^{-3} seconds to send a message. How long will it take the program to run with 1000 processors?

2.11. Derive formulas for the total number of links in the various distributed-memory interconnects.

2.12. **a.** A planar mesh is just like a toroidal mesh, except that it doesn't have the "wraparound" links. What is the bisection width of a square planar mesh?

b. A three-dimensional mesh is similar to a planar mesh, except that it also has depth. What is the bisection width of a three-dimensional mesh?

2.13. **a.** Sketch a four-dimensional hypercube.

b. Use the inductive definition of a hypercube to explain why the bisection width of a hypercube is $p/2$.

2.14. To define the bisection width for indirect networks, the processors are partitioned into two groups so that each group has half the processors. Then, links are removed from *anywhere* in the network so that the two groups are no longer connected. The minimum number of links removed is the bisection width. When we count links, if the diagram uses unidirectional links, two unidirectional links count as one link. Show that an eight-by-eight crossbar has a bisection width less than or equal to eight. Also show that an omega network with eight processors has a bisection width less than or equal to four.

2.15. **a.** Suppose a shared-memory system uses snooping cache coherence and write-back caches. Also suppose that core 0 has the variable `x` in its cache, and it executes the assignment `x = 5`. Finally suppose that core 1 doesn't have `x` in its cache, and after core 0's update to `x`, core 1 tries to execute `y = x`. What value will be assigned to `y`? Why?

b. Suppose that the shared-memory system in the previous part uses a directory-based protocol. What value will be assigned to `y`? Why?

c. Can you suggest how any problems you found in the first two parts might be solved?

2.16. **a.** Suppose the run-time of a serial program is given by $T_{\text{serial}} = n^2$, where the units of the run-time are in microseconds. Suppose that a parallelization of this program has run-time $T_{\text{parallel}} = n^2/p + \log_2(p)$. Write a program that finds the speedups and efficiencies of this program for various values of n and p. Run your program with $n = 10, 20, 40, \ldots, 320$, and $p = 1, 2, 4, \ldots, 128$. What happens to the speedups and efficiencies as p is increased and n is held fixed? What happens when p is fixed and n is increased?

b. Suppose that $T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}$. Also suppose that we fix p and increase the problem size.

- Show that if T_{overhead} grows more slowly than T_{serial}, the parallel efficiency will increase as we increase the problem size.
- Show that if, on the other hand, T_{overhead} grows faster than T_{serial}, the parallel efficiency will decrease as we increase the problem size.

2.17. A parallel program that obtains a speedup greater than p—the number of processes or threads—is sometimes said to have **superlinear speedup**. However, many authors don't count programs that overcome "resource limitations" as having superlinear speedup. For example, a program that must use secondary storage for its data when it's run on a single processor system might be able to fit all its data into main memory when run on a large distributed-memory system. Give another example of how a program might overcome a resource limitation and obtain speedups greater than p.

2.18. Look at three programs you wrote in your Introduction to Computer Science class. What (if any) parts of these programs are inherently serial? Does the inherently serial part of the work done by the program decrease as the problem size increases? Or does it remain roughly the same?

2.19. Suppose $T_{\text{serial}} = n$ and $T_{\text{parallel}} = n/p + \log_2(p)$, where times are in microseconds. If we increase p by a factor of k, find a formula for how much we'll need to increase n in order to maintain constant efficiency. How much should we increase n by if we double the number of processes from 8 to 16? Is the parallel program scalable?

2.20. Is a program that obtains linear speedup strongly scalable? Explain your answer.

2.21. Bob has a program that he wants to time with two sets of data, `input_data1` and `input_data2`. To get some idea of what to expect before adding timing functions to the code he's interested in, he runs the program with two sets of data and the Unix shell command `time`:

```
$ time ./bobs_prog < input_data1

real 0m0.001s
user 0m0.001s
sys  0m0.000s
$ time ./bobs_prog < input_data2

real 1m1.234s
user 1m0.001s
sys  0m0.111s
```

The timer function Bob is using has millisecond resolution. Should Bob use it to time his program with the first set of data? What about the second set of data? Why or why not?

2.22. As we saw in the preceding problem, the Unix shell command `time` reports the user time, the system time, and the "real" time or total elapsed time. Suppose that Bob has defined the following functions that can be called in a C program:

```
double utime(void);
double stime(void);
double rtime(void);
```

The first returns the number of seconds of user time that have elapsed since the program started execution, the second returns the number of system seconds, and the third returns the total number of seconds. Roughly, user time is time spent in the user code and library functions that don't need to use the operating system—for example, `sin` and `cos`. System time is time spent in functions that do need to use the operating system—for example, `printf` and `scanf`.

a. What is the mathematical relation among the three function values? That is, suppose the program contains the following code:

```
u = double utime(void);
s = double stime(void);
r = double rtime(void);
```

Write a formula that expresses the relation between `u`, `s`, and `r`. (You can assume that the time it takes to call the functions is negligible.)

b. On Bob's system, any time that an MPI process spends waiting for messages isn't counted by either `utime` or `stime`, but the time *is* counted by `rtime`. Explain how Bob can use these facts to determine whether an MPI process is spending too much time waiting for messages.

c. Bob has given Sally his timing functions. However, Sally has discovered that on her system, the time an MPI process spends waiting for messages is counted as user time. Furthermore, sending messages doesn't use any system time. Can Sally use Bob's functions to determine whether an MPI process is spending too much time waiting for messages? Explain your answer.

2.23. In our application of Foster's methodology to the construction of a histogram, we essentially identified aggregate tasks with elements of `data`. An apparent alternative would be to identify aggregate tasks with elements of `bin_counts`, so an aggregate task would consist of all increments of `bin_counts[b]` and consequently all calls to `Find_bin` that return `b`. Explain why this aggregation might be a problem.

2.24. If you haven't already done so in Chapter 1, try to write pseudo-code for our tree-structured global sum, which sums the elements of `loc_bin_cts`. First consider how this might be done in a shared-memory setting. Then consider how this might be done in a distributed-memory setting. In the shared-memory setting, which variables are shared and which are private?