UNIT I MULTI-CORE PROCESSORS

Single core to Multi-core architectures – SIMD and MIMD systems – Interconnection networks - Symmetric and Distributed Shared Memory Architectures – Cache coherence - Performance Issues – Parallel program design.

1.1 Introduction

1. From 1986 to 2002 the performance of microprocessors increased, on average, 50% per year.
2. This unprecedented increase meant that users and software developers could often simply wait for the next generation of microprocessors in order to obtain increased performance from an application program.
3. By 2005, rather than trying to continue to develop ever-faster monolithic processors, manufacturers started putting multiple complete processorson a single integrated circuit.
4. This change has a very important consequence for software developers: simply adding more processors will not magically improve the performance of the vast majority of serial programs, that is, programs that were written to run on a single processor.
5. Such programs are unaware of the existence of multiple processors, and the performance of such a program on a system with multiple processors will be effectively the same as its performance on a single processor of the multiprocessor system.

1.2 Single core to Multi-core architectures

1.2.1 Increasing Performance

1. The vast increases in computational power that we've been enjoying for decades now have been at the heart of many of the most dramatic advances in fields as diverse as science, the Internet, and entertainment.
2. As our computational power increases, the number of problems that we can seriously consider solving also increases. The following are a few examples:

Climate modeling.

1. In order to better understand climate change, we need far more accurate computer models, models that include interactions between the atmosphere, the oceans, solid land, and the ice caps at the poles.
2. We also need to be able to make detailed studies of how various interventions might affect the global climate.

Protein folding.

1. It's believed that misfolded proteins may be involved in diseases such as Huntington's, Parkinson's, and Alzheimer's, but our ability to study configurations of complex molecules such as proteins is severely limited by our current computational power.

Drug discovery.

2. There are many ways in which increased computational power can be used in research into new medical treatments.

Energy research.

1. Increased computational power will make it possible to program much more detailed models of technologies such as wind turbines, solar cells, and batteries.
2. These programs may provide the information needed to construct far more efficient clean energy sources.

***Data analysis*.**

1. We generate tremendous amounts of data. By some estimates, the quantity of data stored worldwide doubles every two years , but the vast majority of it is largely useless unless it's analyzed.

These and a host of other problems won't be solved without vast increases in computational power.

1.2.2 Building parallel systems

1. Much of the tremendous increase in single processor performance has been driven by the ever-increasing density of transistors—the electronic switches—on integrated circuits.
2. As the size of transistors decreases, their speed can be increased, and the overall speed of the integrated circuit can be increased.
3. However, as the speed of transistors increases, their power consumption also increases.
4. Most of this power is dissipated as heat, and when an integrated circuit gets too hot, it becomes unreliable.
5. In the first decade of the twenty-first century, air-cooled integrated circuits are reaching the limits of their ability to dissipate heat.
6. We exploit the continuing increase in transistor density? The answer is *parallelism.* Rather than building ever-faster, more complex, monolithic processors, the industry has decided to put multiple, relatively simple, complete processors on a single chip. Such integrated circuits are called **multicore** processors, and **core** has become synonymous with central processing unit, or CPU.
7. In this setting a conventional processor with one CPU is often called a **single-core** system.

1.2.3 Writing parallel programs

1. Most programs that have been written for conventional, single-core systems cannot exploit the presence of multiple cores.
2. We can run multiple instances of a program on a multicore system, but this is often of little help.
3. For example, being able to run multiple instances of our favorite game program isn't really what we want—we want the program to run faster with more realistic graphics.
4. In order to do this, we need to either rewrite our serial programs so that they're *parallel*, so that they can make use of multiple cores, or write translation programs, that is, programs that will automatically convert serial programs into parallel programs.
5. The bad news is that researchers have had very limited success writing programs that convert serial programs in languages such as C and C++ into parallel programs.
6. For example, we can view the multiplication of two *n* x *n* matrices as a sequence of dot products, but parallelizing a matrix multiplication as a sequence of parallel dot products is likely to be very slow on many systems.
7. As an example, suppose that we need to compute *n* values and add them together. We know that this can be done with the following serial code:

```
sum = 0;
for (i = 0; i < n; i++) {
x = Compute_next_value(. . .);
sum += x;
}
```

Now suppose we also have *p* cores and *p* is much smaller than *n*. Then each core can form a partial sum of approximately *n*=*p* values:

```
my_sum = 0;
my_first i = . . . ;
```

```
my_last i = . . . ;
for (my_i = my_first i; my_i < my_last i; my_i++) {
my_x = Compute_next_value(. . .);
        my_sum += my_x;
        }
```

Here the prefix my_ indicates that each core is using its own, private variables, and each core can execute this block of code independently of the other cores.

After each core completes execution of this code, its variable my sum will store the sum of the values computed by its calls to Compute_next_value. For example, if there are eight cores, *n* =24, and the 24 calls to Compute next value return the values

1, 4, 3, 9, 2, 8, 5, 1, 1, 6, 2, 7, 2, 5, 0, 4, 1, 8, 6, 5, 1, 2, 3, 9, then the values stored in my_sum might be

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Here we're assuming the cores are identified by nonnegative integers in the range 0, 1, : : : ,*p*-1, where *p* is the number of cores. When the cores are done computing their values of my_sum, they can form a global sum by sending their results to a designated "master" core, which can add their results:

```
        if (I'm the master core) {
        sum = my_x;
        for each core other than myself {
        receive value from core;
        sum += value;
}
        } else {
        send my_x to the master;
}
```

In our example, if the master core is core 0, it would add the values 8+19+7+15+7+13+12+14 = 95.

1. But you can probably see a better way to do this—especially if the number of cores is large.
2. Instead of making the master core do all the work of computing the final sum, we can pair the cores so that while core 0 adds in the result of core 1, core 2 can add in the result of core 3, core 4 can add in the result of core 5 and so on.
3. Then we can repeat the process with only the even-ranked cores: 0 adds in the result of 2, 4 adds in the result of 6, and so on. Now cores divisible by 4 repeat the process, and so on.

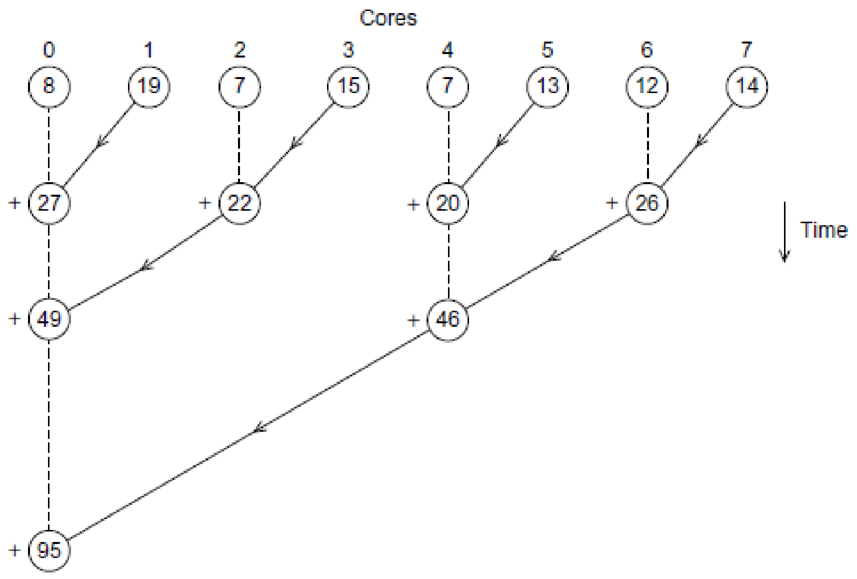


Fig.1.1 Multiple cores forming a global sum

Task-parallelism and Data-parallelism

1. There are two basic ideas of partitioning the work among the cores are task-parallelism and data-parallelism.
2. In task-parallelism, we partition the various tasks carried out in solving the problem among the cores.
3. In data-parallelism, we partition the data used in solving the problem among the cores, and each core carries out more or less similar operations on its part of the data.

MPI, Pthreads and OpenMP

1. We'll be focusing on learning to write programs that are *explicitly* parallel.
2. Our purpose is to learn the basics of programming parallel computers using the C language and three different extensions to C: the **Message-Passing Interface** or **MPI, POSIX threads** or **Pthreads,** and **OpenMP**.
3. MPI and Pthreads are libraries of type definitions, functions, and macros that can be used in C programs.
4. OpenMP consists of a library and some modifications to the C compiler.

1.3 SIMD and MIMD systems

Parallel Hardware

1. Multiple issue and pipelining can clearly be considered to be parallel hardware, since functional units are replicated.
2. 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.

1.3.1 SIMD systems

1. In parallel computing, **Flynn's taxonomy** is frequently used to classify computer architectures.
2. It classifies a system according to the number of instruction streams and the number of data streams it can simultaneously manage.

3. 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.
4. **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.
5. 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.

   (1) 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 *i*th ALU, have the *i*th 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.

   (2) 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.

Finally, as our first example shows, SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data. 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.

Vector processors

1. 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.*
2. 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.
Interleaved memory. The memory system consists of multiple "banks" of memory, which can be accessed more or less independently.
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

3. 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.
4. 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.
5. 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.

Graphics processing units

1. Real-time graphics application programming interfaces, or APIs, use points, lines, and triangles to internally represent the surface of an object.
2. 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.
3. 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.
4. They're also implicitly parallel, since they can be applied to multiple elements (e.g., vertices) in the graphics stream.
5. 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.
6. 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;

1.3.2 MIMD systems

1. **Multiple instruction, multiple data**, or MIMD, systems support multiple simultaneous instruction streams operating on multiple data streams.
2. 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.
3. Furthermore, unlike SIMD systems, MIMD systems are usually **asynchronous**, that is, the processors can operate at their own pace.
4. In many MIMD systems there is no global clock, and there may be no relation between the system times on two different processors.

Shared-memory and distributed-memory system

1. As we noted earlier, there are two principal types of MIMD systems: shared-memory systems and distributed-memory systems.

2. 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.
3. In a shared-memory system, the processors usually communicate implicitly by accessing shared data structures.
4. 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.

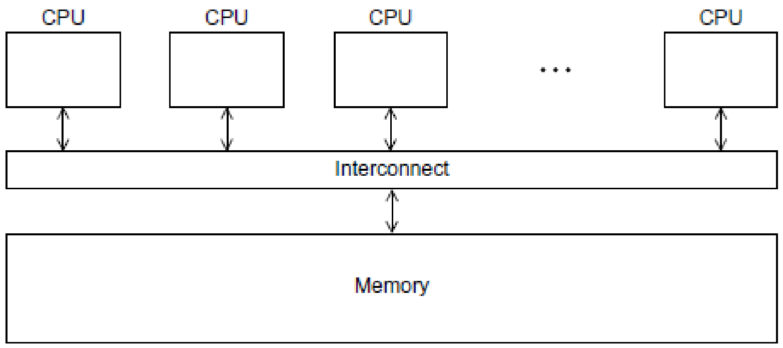


Fig. 1.2 A shared-memory system

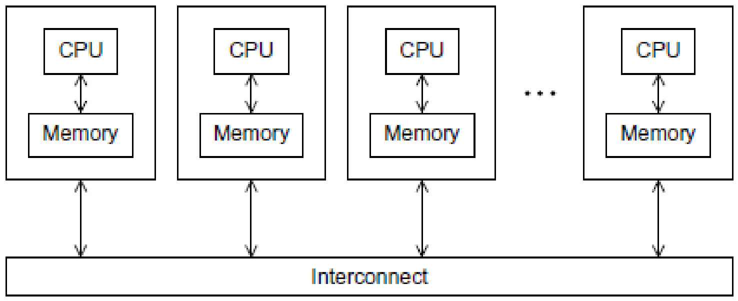


Fig. 1.3 A distributed-memory system

Shared-memory systems

1. The most widely available shared-memory systems use one or more **multicore** processors.
2. 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.

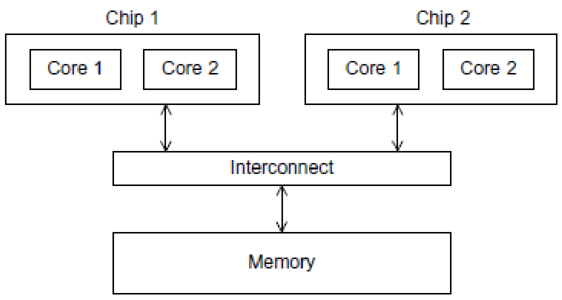


Fig. 1.4 A UMA multicore system

3. 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 1.4 and 1.5.

4. 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.
5. 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.

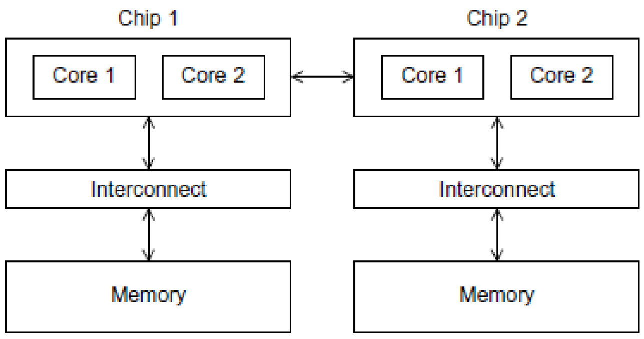


Fig. 1.5 A NUMA multicore system

Distributed-memory systems

1. 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.
2. 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.
3. 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.
4. 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.

1.4 Interconnection networks

1. 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.

Shared-memory interconnects

1. 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.
2. The key characteristic of a bus is that the communication wires are shared by the devices that are connected to it.

3. 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.
4. 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.
5. As the name suggests, **switched** interconnects use switches to control the routing of data among the connected devices. A **crossbar** is illustrated in Figure 1.6 (a). The lines are bidirectional communication links, the squares are cores or memory modules, and the circles are switches.

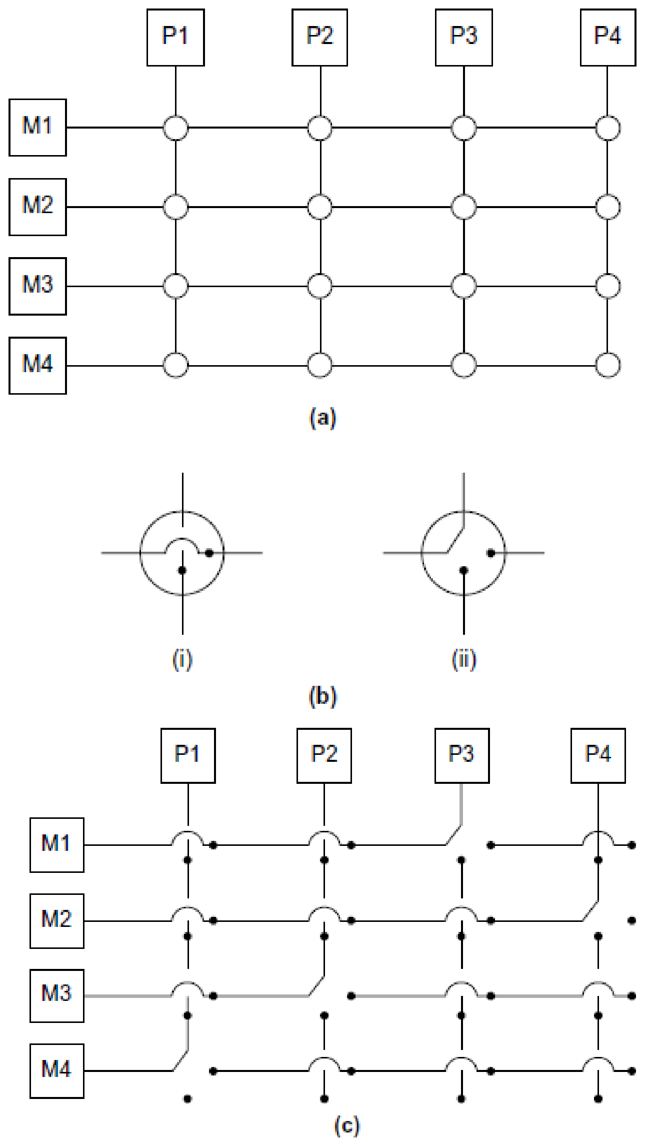


Fig. 1.6 (a) A crossbar switch connecting four processors (*Pi*) and four memory modules (*Mj*); (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors

6. The individual switches can assume one of the two configurations shown in Figure 1.6 (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 if the two cores attempt to simultaneously access the same memory module.
7. For example, Figure 1.6 (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.
8. Crossbars allow simultaneous communication among different devices, so they are much faster than buses.

Distributed-memory interconnects

1. Distributed-memory interconnects are often divided into two groups: direct interconnects and indirect interconnects.
2. In a **direct interconnect** each switch is directly connected to a processor-memory pair, and the switches are connected to each other.
3. Figure 1.7 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.
4. 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 3*p* in a toroidal mesh, while it's only 2*p* in a ring.

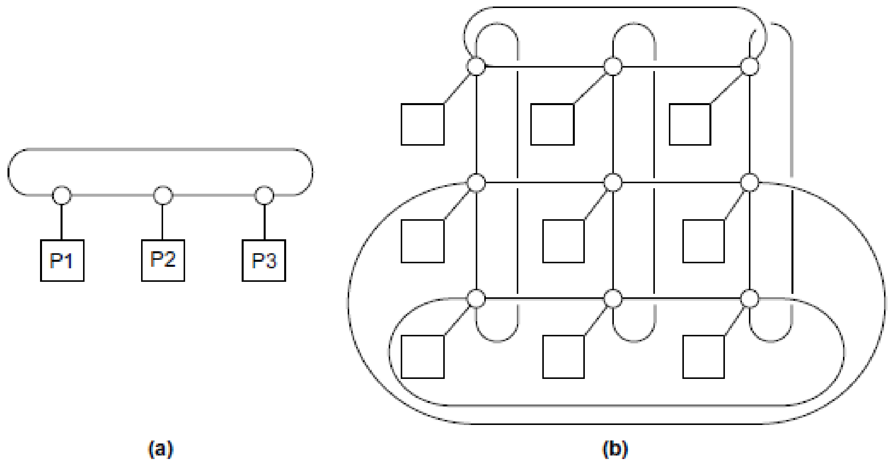


Fig. 1.7 (a) A ring and (b) a toroidal mesh

5. One measure of "number of simultaneous communications" or "connectivity" is **bisection width**.
6. To understand this measure, imagine that the parallel system is 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 1.8 (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.
7. However, in Figure 1.8(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.

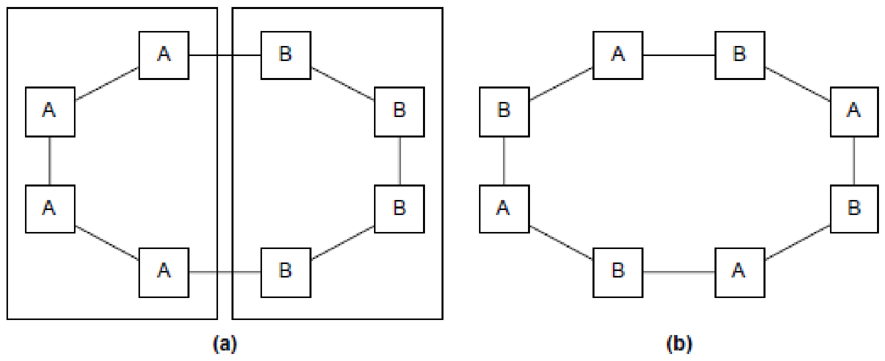


Fig.1.8 Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place

8. The **bandwidth** of a link is the rate at which it can transmit data. It's usually given in megabits or megabytes per second.
9. **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.
10. The ideal direct interconnect is a **fully connected network** in which each switch is directly connected to every other switch. See Figure1.9 . 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 must be capable of connecting to p links.
11. It is therefore more a "theoretical best possible" interconnect than a practical one, and it is used as a basis for evaluating other interconnects.

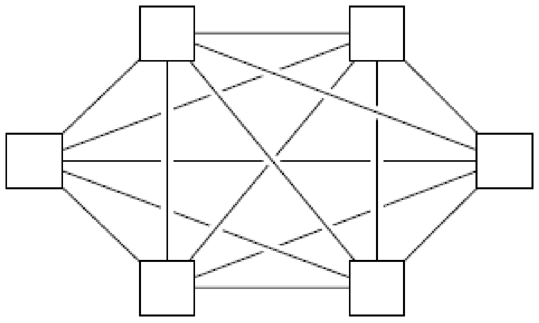


Fig. 1.9 A fully connected network

Hypercube

1. The **hypercube** is a highly connected direct interconnect that has been used in actual systems. Hypercubes are built inductively:
2. A one-dimensional hypercube is a fully-connected system with two processors.
3. 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 1.10.
4. 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.
5. 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.

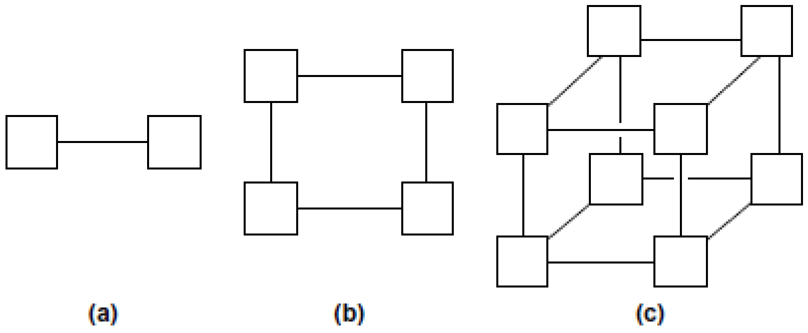


Fig. 1.10 (a) One-, (b) two-, and (c) three-dimensional hypercubes

Indirect interconnects

1. **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.

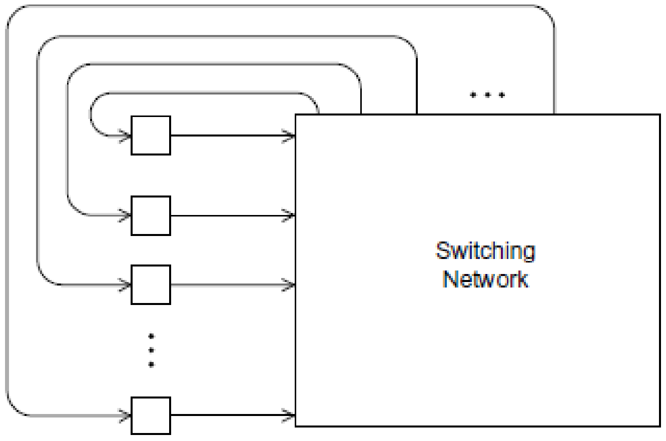


Fig. 1.11A generic indirect network

Crossbar and Omega network

1. The **crossbar** and the **omega network** are relatively simple examples of indirect networks. We saw a shared-memory crossbar with bidirectional links earlier.
2. The diagram of a distributed-memory crossbar in Figure 1.12 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.
3. An omega network is shown in Figure 1.13. The switches are two-by-two crossbars Observe that unlike the crossbar, there are communications that cannot occur simultaneously.
4. For example, in Figure 1.13 if processor 0 sends 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.
5. The omega network uses 1 ½ *p*log2 (p) of the 2x2 crossbar switches, so it uses a total of $2p\log_2$ (p) switches, while the crossbar uses p^2.

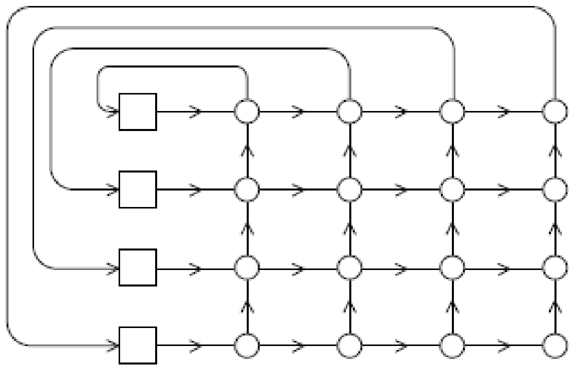


Fig. 1.12 A crossbar interconnect for distributed-memory

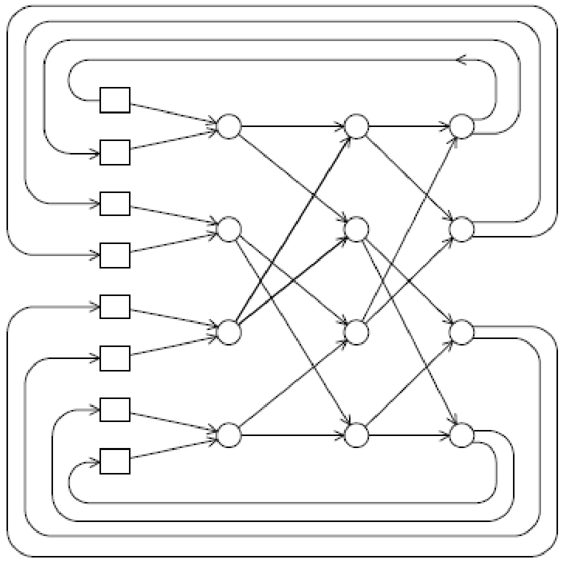


Fig. 1.13 An omega network

Latency and bandwidth

1. 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.
2. 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

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

1.5 Cache coherence

1. 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 1.14.
2. 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 4x7 = 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 4x2 = 8.

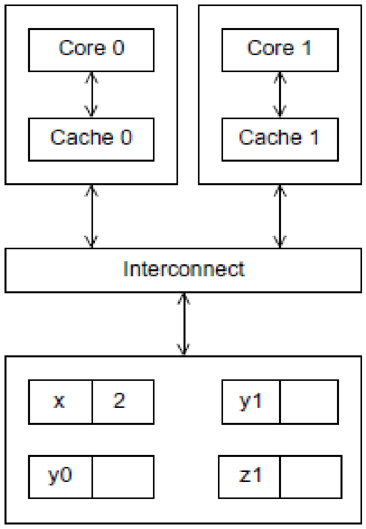


Fig. 1.14 A shared-memory system with two cores and two caches

3. 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.
4. 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.
5. 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

1. There are two main approaches to insuring cache coherence: **snooping cache coherence** and directory-based cache coherence.

2. 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.

3. 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.
4. 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.

Directory-based cache coherence

1. Unfortunately, in large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated. So snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade.
2. 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. Such a system can, in principle, scale to very large numbers of cores.
3. 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.
4. **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.
5. 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.

False sharing

1. 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.
2. 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+1; i < 2*iter_count; i++)
 for (j = 0; j < n; j++)
 y[i] += f(i,j);
   . . .
```

1. 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.
2. 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.

1.6 Shared-memory versus distributed-memory

1. 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.
2. 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.
3. 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.
4. Large crossbars are very expensive, so it's also unusual to find systems with large crossbar interconnects.
5. 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.
6. Thus, distributed-memory systems are often better suited for problems requiring vast amounts of data or computation.

1.6.1 Shared-memory

1. 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.
2. Communication among the threads is usually done through shared variables, so communication is implicit, rather than explicit.

Dynamic and static threads

1. 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.
2. 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.
3. 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.
4. 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.

Nondeterminism

1. In any MIMD system in which the processors execute asynchronously it is likely that there will be **nondeterminism**.
2. 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.
3. 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 = %dnn", 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
```

4. In fact, things could be even worse: the output of one thread could be broken up by the output of the other thread.
5. 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.

Thread safety

1. 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.
2. 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.
3. Thus, static variables are effectively shared among any threads that call the function, and this can have unexpected and unwanted consequences.
4. 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.

5. 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.
6. 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.
7. 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.

1.6.2 Distributed-memory

1. 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.
2. 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

Message-passing

1. 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, … , *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);
}
```

2. 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.
3. 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).
4. 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.
5. 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.

One-sided communication

1. 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.
2. 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.

Partitioned global address space languages

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];
```

1. 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.
2. 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.
3. 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.
4. **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.
5. 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.

1.7 Performance Issues

Speedup and efficiency

1. 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.
2. 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*parallel = *T*serial/*p*. When this happens, we say that our parallel program has **linear speedup**.
3. In practice, we're unlikely to get linear speedup because the use of multiple processes/threads almost invariably introduces some overhead.

4. 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.
5. 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.
6. 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_1 increases. Table 1 shows an example of the changes in S and S/p as p increases. 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

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

Table 1. 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

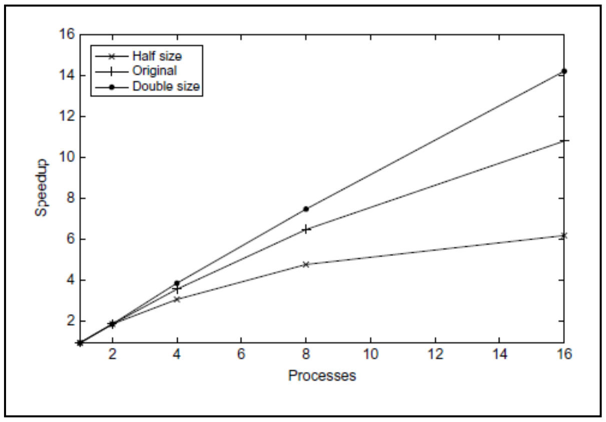
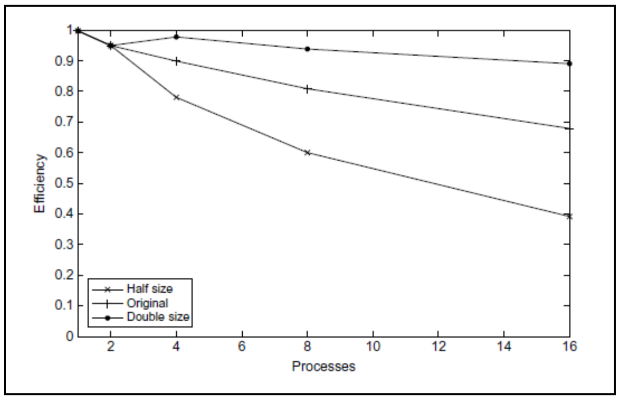


Table 2. Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	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



Amdahl's law

1. Back in the 1960s, Gene Amdahl made an observation 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.
2. 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*.
3. If the serial run-time is *T*serial= 20 seconds, then the run-time of the parallelized part will be 0.9X*T*serial/*p* = 18/*p* and the run-time of the "unparallelized" part will be 0.1X*T*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_*T*serial=*p* D 18=*p* gets closer and closer to 0, so the total parallel run-time can't be smaller than 0.1_*T*serial D 2. That is, the denominator in *S* can't be smaller than 0.1_*T*serial D 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* ≤ 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.

Scalability

1. 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.
2. 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.
3. 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**.
4. As an example, suppose that *T*serial = *n*, where the units of *T*serial are in microseconds, and *n* is also the problem size. Also suppose that *T*parallel = *n*/*p*+1. Then

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

5. 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.
6. 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.

Taking timings

1. There are a *lot* of different approaches to find *T*serial and *T*parallel, and with parallel programs the details may depend on the API.
2. However, there are a few general observations we can make that may make things a little easier.
3. 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.
4. 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.
5. 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.
6. It would include the time for code we've written; 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.
7. 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);
```

8. 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.

1.8 <u>Parallel Program Design</u>

1. To parallelize the serial program, 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.
2. In most cases, we also need to arrange for the processes/threads to synchronize and communicate.
3. 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,
4. However, Ian Foster provides an outline of steps in his online book *Designing and Building Parallel Programs* [19]:
   a. *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.
   b. *Communication*. Determine what communication needs to be carried out among the tasks identified in the previous step.
   c. *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.
   d. *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**.

An example

1. 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.
2. 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 1.15.

A serial program

1. 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.
2. 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;

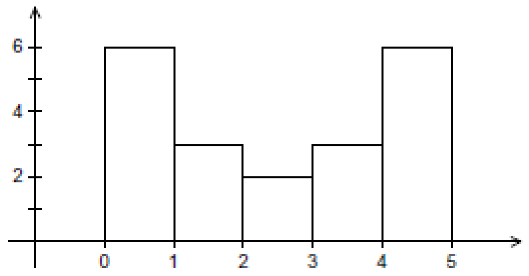


Fig.1.15 A histogram

1. 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
2. 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);

Parallelizing the serial program

1. 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.
2. 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.
3. 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.
4. For the second step, there must be a communication between the computation of the appropriate bin and incrementing an element of bin counts.

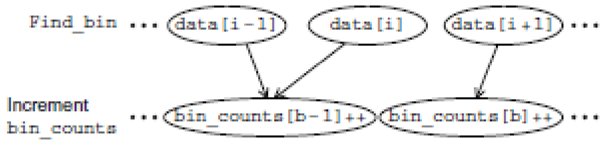


Fig. 1.16 The first two stages of Foster's methodology

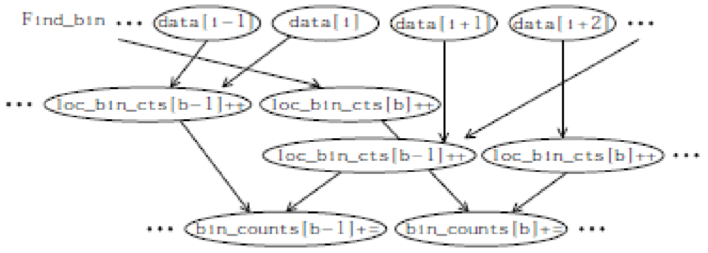


Fig. 1.17 Alternative definition of tasks and communication

Now we 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.

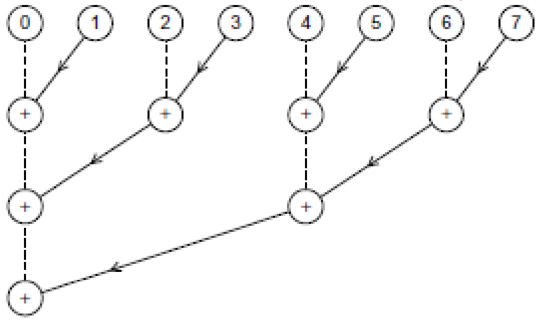


Fig. 1.18 Adding the local arrays

3. Each 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.
4. 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.