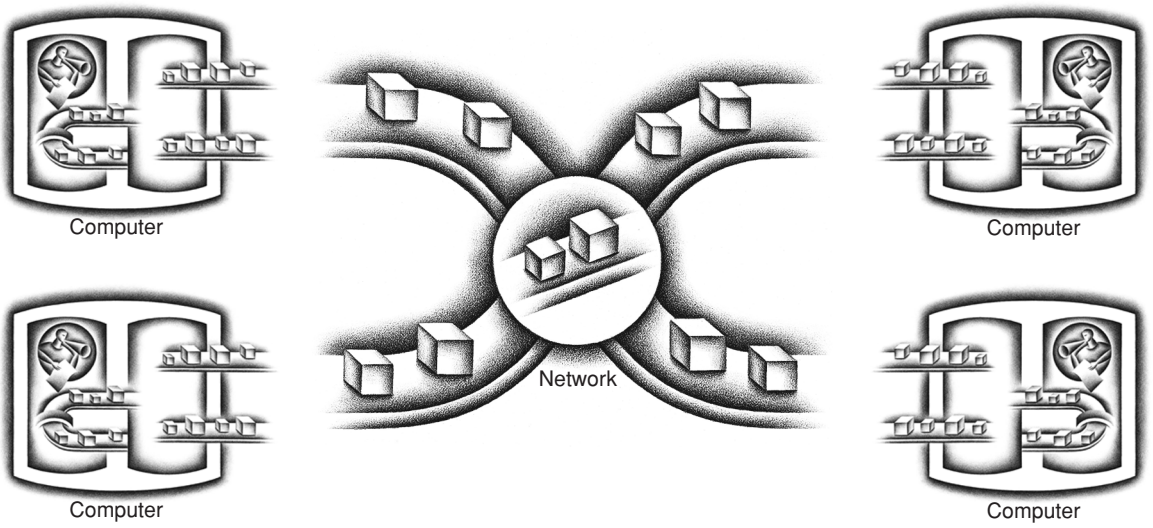
9

Multiprocessors and Clusters

There are finer fish in the sea than have ever been caught.

Irish proverb

The Five Classic Components of a Computer



"Over the Mountains
Of the Moon,
Down the Valley of the
Shadow,
Ride, boldly ride"
The shade replied,—
"If you seek for Eldorado!"

Edgar Allan Poe, "Eldorado," stanza 4, 1849

9.1 Introduction

Computer architects have long sought the El Dorado of computer design: to create powerful computers simply by connecting many existing smaller ones. This golden vision is the fountainhead of **multiprocessors**. The customer orders as many processors as the budget allows and receives a commensurate amount of performance. Thus, multiprocessors must be scalable: the hardware and software are designed to be sold with a variable number of processors, with some machines varying by a factor of more than 50. Since software is scalable, some multiprocessors can support operation in the presence of broken hardware; that is, if a single processor fails in a multiprocessor with n processors, the system provides continued service with $n - 1$ processors. Finally, multiprocessors have the highest absolute performance—faster than the fastest uniprocessor.

multiprocessor Parallel processors with a single shared address.

The good news is that the multiprocessor has established a beachhead. Keeping in mind that the microprocessor is now the most cost-effective processor, it is generally agreed that if you can't handle a workload on a microprocessor, then a multiprocessor or **cluster** composed of many microprocessors is more effective than building a high-performance uniprocessor from a more exotic technology. There are many scientific applications that are too demanding to make progress on them with a single microprocessor: weather prediction, protein folding, and even search for extraterrestrial intelligence. Thus, Figure 9.1.1 shows that the high-performance computing industry depends on multiprocessors and clusters.

cluster A set of computers connected over a local area network (LAN) that function as a single large multiprocessor.

There are also applications outside the sciences that are demanding: search engines, Web servers, and databases. For example, Figure 9.1.2 illustrates that the database industry has standardized on multiprocessors and clusters. Consequently, they now embody a significant market.

Commercial multiprocessors and clusters usually define high performance as high throughput for independent tasks. This definition is in contrast to running a single task on multiple processors. We use the term **parallel processing program** to refer to a single program that runs on multiple processors simultaneously.

parallel processing program A single program that runs on multiple processors simultaneously.

Here are key questions that drive the designs of multiprocessors and clusters:

- How do parallel processors share data?
- How do parallel processors coordinate?
- How many processors?

shared memory A memory for a parallel processor with a single address space, implying implicit communication with loads and stores.

The answers to the first question fall in two main camps. Processors with a *single address space*, sometimes called **shared-memory** processors, offer the programmer a single memory address space that all processors share. Processors

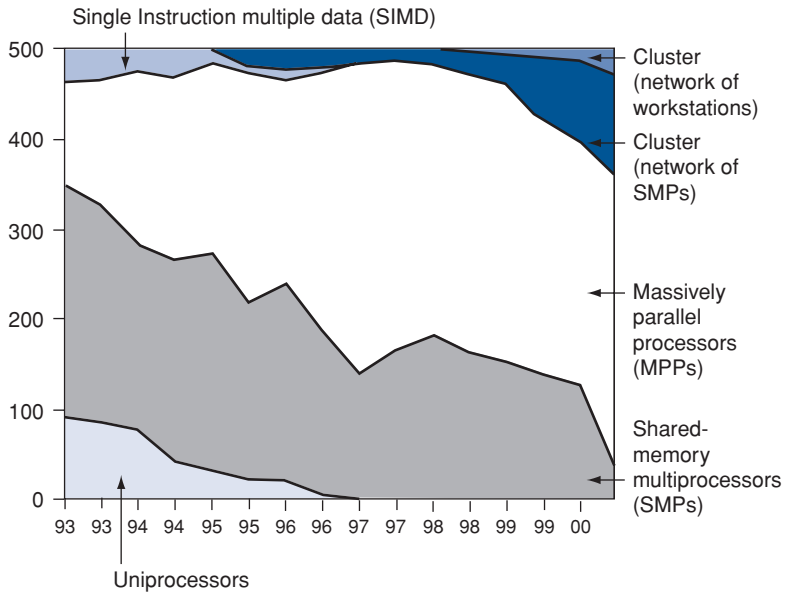


FIGURE 9.1.1 Plot of top 500 supercomputer sites over a decade. The numbers for 1993/1998/2003 are 93/0/0 for uniprocessor, 251/175/0 for SMP, 121/310/165 for MPP, 35/0/0 for SIMD, 0/14/127 for cluster of SMPs, and 0/1/208 for cluster of workstations. Note that in the last five years uniprocessors, SMPs, and SIMDs have disappeared while clusters of various kinds grew from 3% to 67%. Moreover, most of the MPPs in the list look similar to clusters. Performance is measured as the speed of running Linpack, which solves a dense system of linear equations. This list at *www.top500.org* is updated twice a year. This site uses the term *constellation* to mean a network of SMP servers and the term *cluster* to mean a cluster of PCs or workstations, which can be either uniprocessors or small SMPs. This vague distinction is not used in this text; in this book, a cluster is a collection of computers connected by a standard LAN that is used for a common task.

communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data; otherwise, one processor could start working on data before another is finished with it. This coordination is called **synchronization**. When sharing is supported with a single address space, there must be a separate mechanism for synchronization. One approach uses a **lock**: only one processor at a time can acquire the lock, and other processors interested in shared data must wait until the original processor unlocks the variable. Locking is described in Section 9.3.

synchronization The process of coordinating the behavior of two or more processes, which may be running on different processors.

lock A synchronization device that allows access to data to only one processor at a time.

Single address space multiprocessors come in two styles. The first takes the same time to access main memory no matter which processor requests it and no

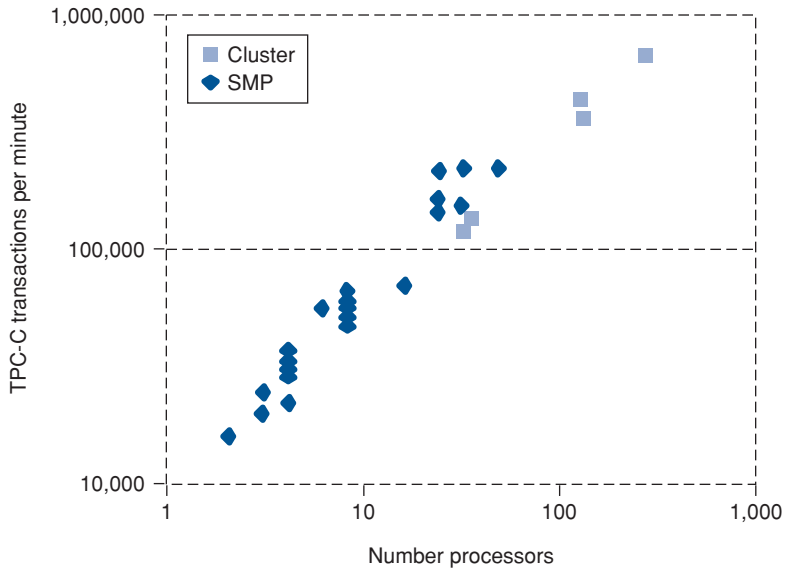


FIGURE 9.1.2 Performance versus number of processor for TPC-C on a log-log scale. These plots are for computers running version 5 of the TPC-C benchmark. Note that even the smallest computer has 2 processors. Clusters get high performance by scaling. They can sustain 2500–3800 transactions per minute per processor from 32 to 280 processors. Not only do clusters have the highest tpmC rating, they have better cost-performance ($/tpmC) than any SMP with a total cost over $1 million.

symmetric multiprocessor (SMP) or **uniform memory access (UMA)** A multiprocessor in which accesses to main memory take the same amount of time no matter which processor requests the access and no matter which word is asked.

nonuniform memory access (NUMA) A type of single-address space multiprocessor in which some memory accesses are faster than others depending which processor asks for which word.

message passing Communicating between multiple processors by explicitly sending and receiving information.

send message routine A routine used by a processor in machines with private memories to pass to another processor.

receive message routine A routine used by a processor in machines with private memories to accept a message from another processor.

matter which word is requested. Such machines are called **uniform memory access (UMA)** multiprocessors or **symmetric multiprocessors (SMP)**. In the second style, some memory accesses are faster than others depending on which processor asks for which word. Such machines are called **nonuniform memory access (NUMA)** multiprocessors. As you might expect, the programming challenges are different for a NUMA multiprocessor versus a UMA multiprocessor, but NUMA machines can scale to larger sizes and hence are potentially higher performance. Figure 9.1.3 shows the number of processors and nonlocal memory access times for commercial SMPs and NUMAs.

The alternative model for communicating uses **message passing** for communicating among processors. Message passing is required for machines with *private memories*, in contrast to shared memory. One example is a cluster, which processors in different desktop computers communicate by passing messages over a local area network. Provided the system has routines to **send** and **receive messages**, coordination is built in with message passing since one processor knows when a message is sent, and the receiving processor knows when a message arrives. The receiving processor can then send a message back to the sender saying the message has arrived if the sender needs that confirmation.

Multiprocessor	Year shipped	SMP or NUMA	Maximum processors	Interconnection network	Typical remote memory access time (ns)
Sun Starfire servers	1996	SMP	64	multiple address buses, data switch	500
SGI Origin 3000	1999	NUMA	512	fat hypercube	500
Cray T3E	1996	NUMA	2048	2-way 3D torus	300
HP V series	1998	SMP	32	8 × 8 crossbar	1000
Compaq AlphaServer GS	1999	SMP	32	switched buses	400
Sun V880	2002	SMP	8	switched buses	240
HP Superdome 9000	2003	SMP	64	switched buses	275

FIGURE 9.1.3 Typical remote access times to retrieve a word from a remote memory in shared-memory multiprocessors.

In addition to two main communication styles, multiprocessors are constructed in two basic organizations: processors connected by a single bus, and processors connected by a network. The number of processors in the multiprocessor has a lot to do with this choice. We will examine these two styles in detail in Sections 9.3 and 9.4.

Let's start by looking at the general issues in programming multiprocessors.

The BIG Picture

Figure 9.1.4 shows the relationship between the number of processors in a multiprocessor and choice of shared address versus message-passing communication and the choice of bus versus network physical connection. Shared address is further divided between uniform and nonuniform memory access. Although there are many choices for some numbers of processors, for other regions there is widespread agreement.

Category	Choice		Number of processors
Communicationmodel	Message passing		8–2048
	Shared address	NUMA	8–256
		UMA	2–64
Physical connection	Network		8–256
	Bus		2–36

FIGURE 9.1.4 Options in communication style and physical connection for multiprocessors as the number of processors varies. Note that the shared address space is divided into uniform memory access (UMA) and nonuniform memory access (NUMA) machines.

A major concern which is frequently voiced in connection with very fast computing machines . . . is that they will . . . run out of work. . . . It must be considered that . . . [past] problem size was dictated by the speed of the computing machines then available. . . . For faster machines, the same automatic mechanism will exert pressure towards problems of larger size.

John von Neumann, address presented at IBM seminar on scientific computation, November 1949

9.2 Programming Multiprocessors

The bad news is that it remains to be seen how many important applications will run faster on multiprocessors via parallel processing. The obstacle is not the price of the uniprocessor used to compose multiprocessors, the flaws in topologies of interconnection networks, or the unavailability of appropriate programming languages; the difficulty has been that too few important application programs have been rewritten to complete tasks sooner on multiprocessors. Because it is even harder to find applications that can take advantage of many processors, the challenge is greater for large-scale multiprocessors.

Because of the programming difficulty, most parallel processing success stories are a result of software wizards developing a parallel subsystem that presents a sequential interface. Examples include databases, file servers, computer-aided design packages, and multiprocessing operating systems.

However, why is this so? Why should parallel processing programs be so much harder to develop than sequential programs?

The first reason is that you *must* get good performance and efficiency from the parallel program on a multiprocessor; otherwise, you would use a uniprocessor, as programming is easier. In fact, uniprocessor design techniques such as superscalar and out-of-order execution take advantage of instruction-level parallelism, normally without involvement of the programmer. Such innovation reduces the demand for rewriting programs for multiprocessors.

Why is it difficult to write multiprocessor programs that are fast, especially as the number of processors increases? As an analogy, think of the communication overhead for a task done by one person compared to the overhead for a task done by a committee, especially as the size of the committee increases. Although n peo-

ple may have the potential to finish any task n times faster, the communication overhead for the group may prevent it; n-fold speedup becomes especially unlikely as n increases. (Imagine the change in communication overhead if a committee grows from 10 people to 1000 people to 1,000,000.)

Another reason why it is difficult to write parallel processing programs is that the programmer must know a good deal about the hardware. On a uniprocessor, the high-level language programmer writes the program largely ignoring the underlying machine organization—that's the job of the compiler. Alas, it's not that simple for multiprocessors.

Although this second obstacle is beginning to lessen, our discussion in Chapter 4 reveals a third obstacle: Amdahl's law. It reminds us that even small parts of a program must be parallelized to reach their full potential; thus coming close to linear speedup involves discovering new algorithms that are inherently parallel.

Speedup Challenge

EXAMPLE

Suppose you want to achieve linear speedup with 100 processors. What fraction of the original computation can be sequential?

ANSWER

Amdahl's law (page 267) says,

Execution time after improvement =

$$\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

Substituting for the goal of linear speedup with 100 processors means the execution time is reduced by 100:

$$\frac{\text{Execution time before improvement}}{100} =$$

$$\frac{\text{Execution time affected by improvement}}{100} + \text{Execution time unaffected}$$

Since

Execution time before improvement =

Execution time affected by improvement + Execution time unaffected

if we substitute this in the equation above, we get

$$\frac{\text{Execution time affected by improvement} + \text{Execution time unaffected}}{100}$$

$$= \frac{\text{Execution time affected by improvement}}{100} + \text{Execution time unaffected}$$

Simplifying, we get

$$\frac{\text{Execution time unaffected by improvement}}{100} = \text{Execution time unaffected}$$

This can only be true if Execution time unaffected is 0.

Accordingly, to achieve linear speedup with 100 processors, *none* of the original computation can be sequential. Put another way, to get a speedup of 99 from 100 processors means the percentage of the original program that was sequential would have to be 0.01% or less.

Yet, there are applications with substantial parallelism.

EXAMPLE

Speedup Challenge, Bigger Problem

Suppose you want to perform two sums: one is a sum of two scalar variables and one is a matrix sum of a pair of two-dimensional arrays, size 1000 by 1000. What speedup do you get with 1000 processors?

ANSWER

If we assume performance is a function of the time for an addition, t, then there is 1 addition that does not benefit from parallel processors and 1,000,000 additions that do. If the time before is $1{,}000{,}001\,t$,

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

$$\text{Execution time after improvement} = \frac{1{,}000{,}000\,t}{1000} + 1\,t = 1001$$

Speedup is then

$$\text{Speedup} = \frac{1{,}000{,}001}{1001} = 999$$

Even if the sequential portion expanded to 100 sums of scalar variables versus one sum of a pair of 1000 by 1000 arrays, the speedup would still be 909.

9.3 Multiprocessors Connected by a Single Bus

The high performance and low cost of the microprocessor inspired renewed interest in multiprocessors in the 1980s. Several microprocessors can usefully be placed on a common bus for several reasons:

- Each microprocessor is much smaller than a multichip processor, so more processors can be placed on a bus.
- Caches can lower bus traffic.
- Mechanisms were invented to keep caches and memory consistent for multiprocessors, just as caches and memory are kept consistent for I/O, thereby simplifying programming.

Figure 9.3.1 is a drawing of a generic single-bus multiprocessor.

Traffic per processor and the bus bandwidth determine the useful number of processors in such a multiprocessor. The caches replicate data in their faster memories both to reduce the latency to the data *and* to reduce the memory traffic on the bus.

Parallel Program (Single Bus)

EXAMPLE

Suppose we want to sum 100,000 numbers on a single-bus multiprocessor computer. Let's assume we have 100 processors.

ANSWER

The first step again would be to split the set of numbers into subsets of the same size. We do not allocate the subsets to a different memory, since there is a single memory for this machine; we just give different starting addresses to each processor. `Pn` is the number of the processor, between 0 and 99. All processors start the program by running a loop that sums their subset of numbers:

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
   sum[Pn] = sum[Pn] + A[i]; /* sum the assigned areas*/
```

The next step is to add these many partial sums, so we divide to conquer. Half of the processors add pairs of partial sums, then a quarter add pairs of the new partial sums, and so on until we have the single, final sum. We want each processor to have its own version of the loop counter variable `i`, so we must indicate that it is a "private" variable.

In this example, the two processors must synchronize before the "consumer" processor tries to read the result from the memory location written by the "producer" processor; otherwise, the consumer may read the old value of the data. Here is the code (`half` is private also):

```
half = 100; /* 100 processors in multiprocessor*/
repeat
        synch(); /* wait for partial sum completion*/
        if (half%2 != 0 && Pn == 0)
              sum[0] = sum[0] + sum[half-1];
              /* Conditional sum needed when half is
              odd; Processor0 gets missing element */
        half = half/2; /* dividing line on who sums */
        if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* exit with final sum in Sum[0] */
```

cache coherency Consistency in the value of data between the versions in the caches of several processors.

Recall from Chapter 8 that I/O can experience inconsistencies in the value of data between the version in memory and the version in the cache. This **cache coherence** problem applies to multiprocessors as well as I/O. Unlike I/O, which rarely uses multiple data copies (a situation to be avoided whenever possible), as the second half of the example suggests, multiple processors routinely require copies of the same data in multiple caches. Alternatively, accesses to shared data could be forced always to go around the cache to memory, but that would be too slow and it would require too much bus bandwidth; performance of a multiprocessor program depends on the performance of the system when sharing data.

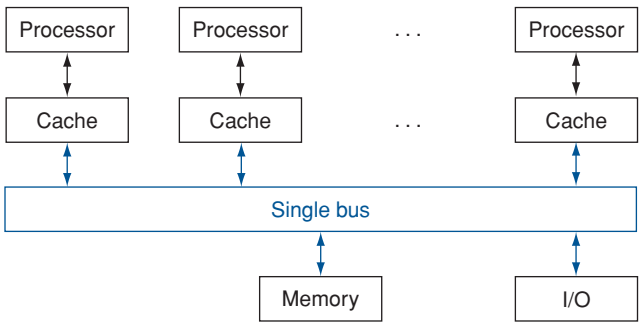


FIGURE 9.3.1 A single-bus multiprocessor. Typical size is between 2 and 32 processors.

The protocols to maintain coherence for multiple processors are called *cache coherence protocols*. The next few subsections explain cache coherence protocols and methods of synchronizing processors using cache coherency.

Multiprocessor Cache Coherence

snooping cache coherency A method for maintaining cache coherency in which all cache controllers monitor or snoop on the bus to determine whether or not they have a copy of the desired block.

The most popular protocol to maintain cache coherence is called **snooping**. Figure 9.3.2 shows how caches access memory over a common bus. All cache controllers monitor, or *snoop*, on the bus to determine whether they have a copy of the shared block.

Snooping became popular with machines of the 1980s, which used single buses to their main memories. These uniprocessors were extended by adding multiple processors on that bus to give easy access to the shared memory. Caches were then added to improve the performance of each processor, leading to schemes to keep the caches up-to-date by snooping on the information over that shared bus.

Maintaining coherence has two components: reads and writes. Multiple copies are not a problem when reading, but a processor must have exclusive access to write a word. Processors must also have the most recent copy when reading an object, so all processors must get new values after a write. Thus, snooping protocols must locate all the caches that share an object to be written. The consequence of a write to shared data is either to invalidate all other copies or to update the shared copies with the value being written.

The status bits already in a cache block are expanded for snooping protocols, and that information is used in monitoring bus activities. On a read miss, all caches check to see if they have a copy of the requested block and then take the appropriate action, such as supplying the data to the cache that missed. Similarly, on a write, all caches check to see if they have a copy and then act, either invalidating or updating their copy to the new value.

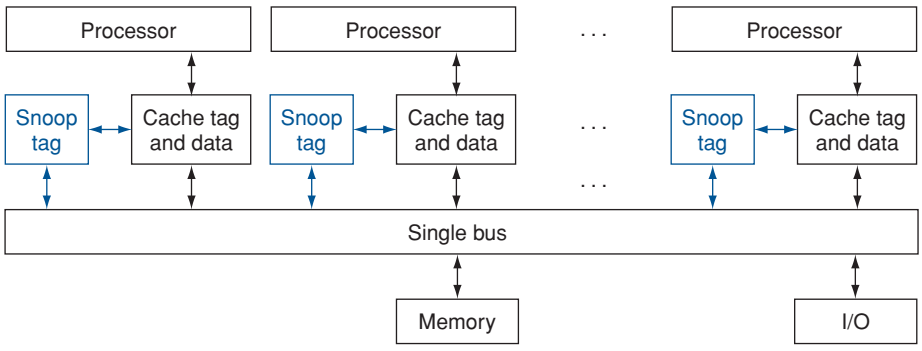


FIGURE 9.3.2 A single-bus multiprocessor using snooping cache coherency. The extra set of tags, shown in color, is used to handle snoop requests. The tags are duplicated to reduce the demands of snooping on the caches.

Since every bus transaction checks cache address tags, you might assume that it interferes with the processor. It would interfere if not for duplicating the address tag portion of the cache—not the whole cache—to get an extra read port for snooping (see Figure 9.3.2). This way, snooping rarely interferes with the processor's access to the cache. When there is interference, the processor will likely stall because the cache is unavailable.

Commercial cache-based multiprocessors use write-back caches because write-back reduces bus traffic and thereby allows more processors on a single bus. To preserve that precious communications bandwidth, all commercial machines use **write-invalidate** as the standard coherence protocol: the writing processor causes all copies in other caches to be invalidated before changing its local copy; it is then free to update the *local* data until another processor asks for it. The writing processor issues an invalidation signal over the bus, and all caches check to see if they have a copy; if so, they must invalidate the block containing the word. Thus, this scheme allows multiple readers but only a single writer.

write-invalidate A type of snooping protocol in which the writing processor causes all copies in other caches to be invalidated before changing its local copy, which allows it to update the local data until another processor asks for it.

Measurements to date indicate that shared data has lower spatial and temporal locality than other types of data. Thus, shared data misses often dominate cache behavior, even though they may be just 10% to 40% of the data accesses. Figure 9.3.3 shows the fraction of misses due to coherence as the number of processors varies in an SMP.

Hardware Software Interface

One insight is that block size plays an important role in cache coherency. For example, take the case of snooping on a cache with a block size of eight words, with a single word alternatively written and read by two processors. A protocol that only broadcasts or sends a single word has an advantage over one that transfers the full block.

Large blocks can also cause what is called **false sharing**: When two unrelated shared variables are located in the same cache block, the full block is exchanged between processors even though the processors are accessing different variables (see Exercises 9.5 and 9.6). Compiler research is under way to reduce false sharing by allocating highly correlated data to the same cache block and thereby reduce cache miss rates.

false sharing A sharing situation in which two unrelated shared variables are located in the same cache block and the full block is exchanged between processors even though the processors are accessing different variables.

Elaboration: In a multiprocessor using cache coherence over a single bus, what happens if two processors try to write to the same shared data word in the same clock cycle? The bus arbiter decides which processor gets the bus first, and this processor will invalidate or update the other processor's copy, depending on the protocol. The second processor then does its write. Bus arbitration forces sequential behavior from writes to the same block by different processors, and this explains how writes from different processors to different words in the same block will work correctly.

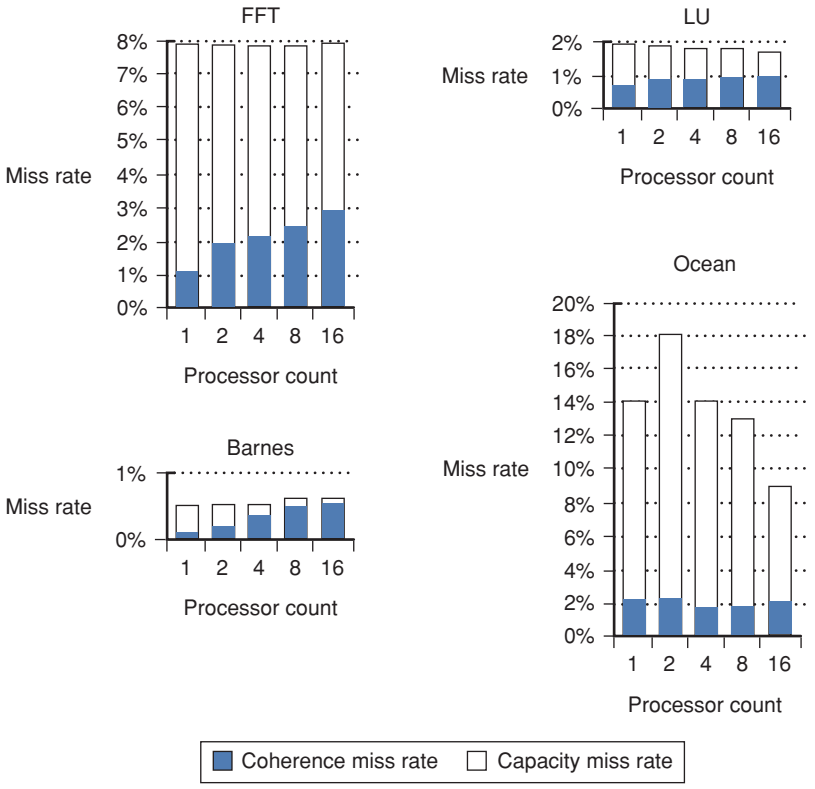


FIGURE 9.3.3 Data miss rates can vary in nonobvious ways as the processor count is increased from 1 to 16. The miss rates include both coherence and capacity miss rates. The compulsory misses in these benchmarks are all very small and are included in the capacity misses. For all these runs, the cache size is 64 KB, two-way set associative, with 32-byte blocks. Notice that the scale on the *y*-axis for each benchmark is different, so that the behavior of the individual benchmarks can be seen clearly. (From Figure 6.23 on page 572 in Hennessy and Patterson, *Computer Architecture: A Quantitative Approach,* third edition, 2003.)

The policy of when a processor sees a write from another processor is called the *memory consistency model*. The most conservative is called *sequential consistency:* the result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved. Some machines use more liberal models to achieve higher memory performance.

Elaboration: Our example used a barrier synchronization primitive; processors wait at the barrier until every processor has reached it. Then they proceed. Barrier synchronization allows all processors to rapidly synchronize. This function can be implemented in software or with the lock synchronization primitive, described shortly.

barrier synchronization A synchronization scheme in which processors wait at the barrier and do not proceed until every processor has reached it.

An Example of a Cache Coherence Protocol

To illustrate the intricacies of a cache coherence protocol, Figure 9.3.4 shows a finite state transition diagram for a write-invalidation protocol based on a write-back policy. Each cache block is in one of three states:

1. *Shared* (read only): This cache block is clean (not written) and may be shared.
2. *Modified* (read/write): This cache block is dirty (written) and may *not* be shared.
3. *Invalid:* This cache block does not have valid data.

The three states of the protocol are duplicated in the figure to show transitions based on processor actions as opposed to transitions based on bus operations. This duplication is done only for purposes of illustration; there is really only one finite state machine per cache block, with stimuli coming either from the attached processor or from the bus. This abstraction applies to caches blocks *not* resident in the case as well; these state machines are obviously all in the invalid state.

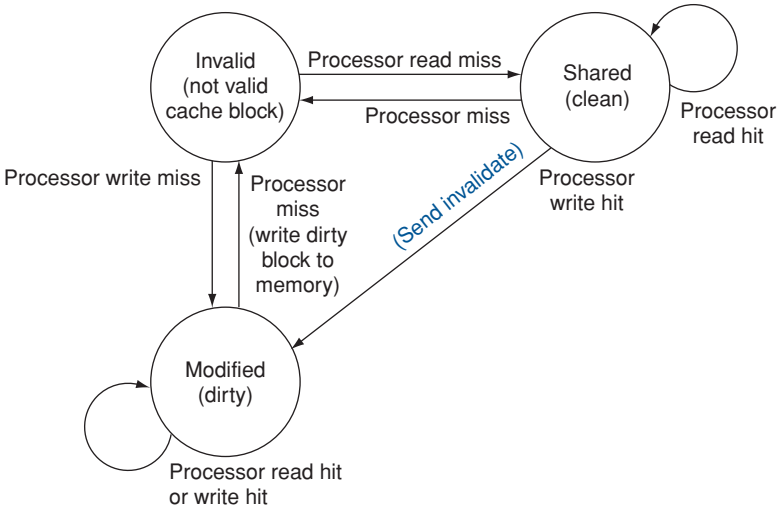
Transitions in the state of a cache block happen on read misses, write misses, or write hits; read hits do not change cache state. Let's start with a read miss. Let's call the block to be replaced the *victim.* When the processor has a read miss, it will acquire the bus, and write back the victim if it was in the Modified state (dirty). All the caches in the other processors monitor the read miss to see if this block is in their cache. If one has a copy and it is in the Modified state, then the block is written back and its state is changed to the Invalid state. (Some protocols would change the state to Shared.) The read miss is then satisfied by reading from memory, and the state of the block is set to Shared. Note that the block is read from memory whether a copy is in a cache or not in this protocol.

Writes are more complex. Let's try write hits. A write hit to a Modified block cause no protocol action. A write hit to a Shared block causes the cache to acquire the bus, send an invalidate signal to knock out any other copies, modify the portion of the block being written, and change the state to Modified.

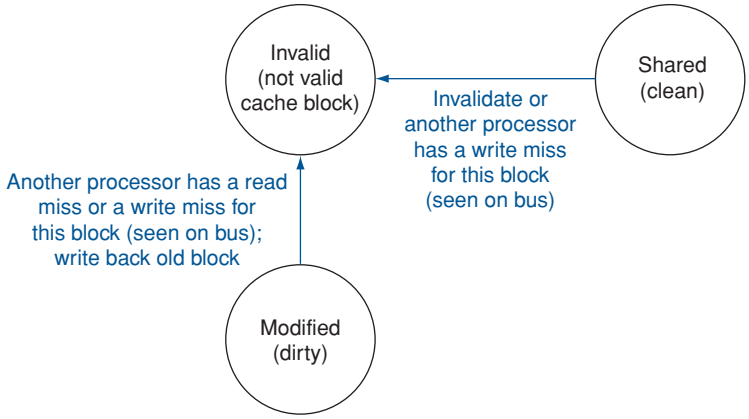
Last is write misses. A write miss to an Invalid block causes the cache to acquire the bus, read the full missing block, modify the portion of the block being written, and change the state to Modified. A write miss to a Shared block in another cache causes the cache to acquire the bus, send an invalidate signal to knock out all copies, read the full missing block, modify the portion of the block being written, and change the state to Modified.

MESI cache coherency protocol A write-invalidate protocol whose name is an acronym for the four states of the protocol: Modified, Exclusive, Shared, Invalid.

As you might imagine, there are many variations on cache coherence that are much more complicated than this simple model. The one found on the Pentium 4 and many other microprocessors is called **MESI**, a write-invalidate protocol whose name is an acronym for the four states of the protocol: Modified, Exclusive,



a. Cache state transitions using signals from the processor



b. Cache state transitions using signals from the bus

FIGURE 9.3.4 A write-invalidate cache coherence protocol. The upper part of the diagram shows state transitions based on actions of the processor associated with this cache; the lower part shows transitions based on actions of other processors as seen as operations on the bus. There is really only one state machine in a cache block, although there are two represented here to clarify when a transition occurs. The black arrows and actions specified in black text would be found in caches without coherency; the colored arrows and actions are added to achieve cache coherency.

Shared, Invalid. Modified and Invalid are the same as above. The Shared state of Figure 9.3.4 is divided, depending on whether there are multiple copies (Shared state) or there is just one (Exclusive state). In either case, memory has an up-to-date version of the data. This extra Exclusive state means there is only one copy of the block, so a write hit doesn't need to invalidate. A write hit to a Shared block in Figure 9.3.4 requires an invalidation, since there may be multiple copies.

Other variations on coherence protocols include whether the other caches try to supply the block if they have a copy, and whether the block must be invalidated on a read miss.

Synchronization Using Coherency

One of the major requirements of a single-bus multiprocessor is to be able to coordinate processes that are working on a common task. Typically, a programmer will use *lock variables* (also known as *semaphores*) to coordinate or synchronize the processes. The challenge for the architect of a multiprocessor is to provide a mechanism to decide which processor gets the lock and to provide the operation that locks a variable. Arbitration is easy for single-bus multiprocessors, since the bus is the only path to memory: the processor that gets the bus locks out all other processors from memory. If the processor and bus provide an **atomic swap operation**, programmers can create locks with the proper semantics. Here the adjective *atomic* means indivisible, so an atomic swap means the processor can both read a location *and* set it to the locked value in the same bus operation, preventing any other processor or I/O device from reading or writing memory until the swap completes.

atomic swap operation An operation in which the processor can both read a location and write it in the same bus operation, preventing any other processor or I/O device from reading or writing memory until

Figure 9.3.5 shows a typical procedure for locking a variable using an atomic swap instruction. Assume that 0 means unlocked ("go") and 1 means locked ("stop"). A processor first reads the lock variable to test its state. A processor keeps reading and testing until the value indicates that the lock is unlocked. The processor then races against all other processors that were similarly *spin waiting* to see who can lock the variable first. All processors use an atomic swap instruction that reads the old value and stores a 1 ("stop") into the lock variable. The single winner will see the 0 ("go"), and the losers will see a 1 that was placed there by the winner. (The losers will continue to write the variable with the locked value of 1, but that doesn't change its value.) The winning processor then executes the code that updates the shared data. When the winner exits, it stores a 0 ("go") into the lock variable, thereby starting the race all over again.

MIPS does not include an atomic swap instruction. An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed *as if* the instructions were atomic. The pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair. Thus, when an instruction pair is effectively atomic, no other processor can change the value between the instruction pair.

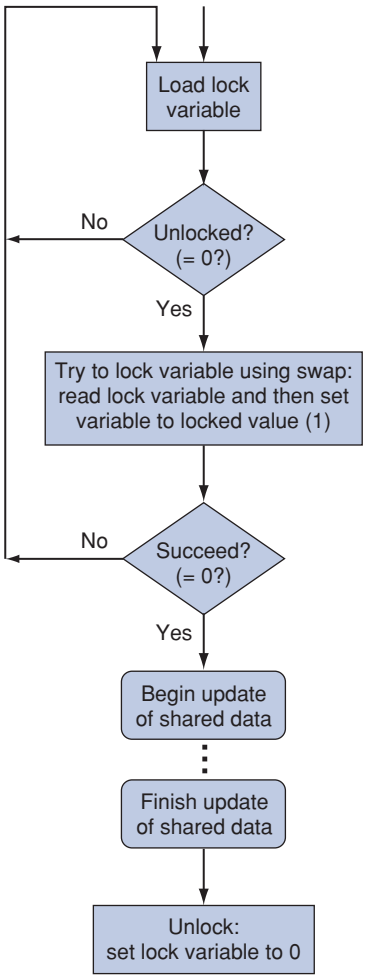


FIGURE 9.3.5 Steps to acquire a lock or semaphore to synchronize processes and then to release the lock on exit from the key section of code.

The MIPS pair of instructions includes a special load called a *load linked* or *load locked* (`ll`) and a special store called a *store conditional* (`sc`). These instructions are used in sequence: If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. If the processor does a context switch between the two instructions, then the store conditional also fails. The store conditional is defined to return a value indicating whether the store was successful. Since the load linked returns the initial value and the store conditional returns 1 if it

succeeds and 0 otherwise, the following sequence implements an atomic exchange on the memory location specified by the contents of $t1:

```
try:  mov   $t3,$t4    # move exchange value
      ll    $t2,0($t1) # load linked
      sc    $t3,0($t1) # store conditional changes $t3
      beqz  $t3,try    # branch if store cond fails (=0)
      nop              # (delayed branch)
      mov   $t4,$t2    # put load value into $t4
```

At the end of this sequence the contents of $t4 and the memory location specified by $t1 have been atomically swapped. Any time a processor intervenes and modifies the value in memory between the ll and sc instructions, the sc returns 0 in $t3, causing the code sequence to try again.

Let's examine how the spin lock scheme of Figure 9.3.5 works with bus-based cache coherency. One advantage of this algorithm is that it allows processors to spin wait on a local copy of the lock in their caches. This reduces the amount of bus traffic; Figure 9.3.6 shows the bus and cache operations for multiple processors trying to lock a variable. Once the processor with the lock stores a 0 into the lock, all other caches see that store and invalidate their copy of the lock variable. Then they try to get the new value for the lock of 0. This new value starts the race to see who can set the lock first. The winner gets the bus and stores a 1 into the lock; the other caches replace their copy of the lock variable containing 0 with a 1. This value indicates the variable is already locked, so they must return to testing and spinning.

This scheme has difficulty scaling up to many processors because of the communication traffic generated when the lock is released.

9.4 Multiprocessors Connected by a Network

Single-bus designs are attractive, but limited because the three desirable bus characteristics are incompatible: high bandwidth, low latency, and long length. There is also a limit to the bandwidth of a single memory module attached to a bus. Thus, a single bus imposes practical constraints on the number of processors that can be connected to it. To date, the largest number of processors connected to a single bus in a commercial computer is 36, and this number seems to be dropping over time.

If the goal is to connect many more processors together, then the computer designer needs to use more than a single bus. Figure 9.4.1 shows how this can be organized. Note that in Figure 9.3.1 on page 9-12, the connection medium—the bus—is between the processors and memory, whereas in Figure 9.4.1, memory is

Step	Processor P0	Processor P1	Processor P2	Bus activity	Memory
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	None	
2	Sets lock to 0; sends invalidate over bus	Spins, testing if lock = 0	Spins, testing if lock = 0	Write-invalidate of lock variable sent from P0	
3		Cache miss	Cache miss	Bus services P2's cache miss	
4	Responds to P2's cache miss; sends lock = 0	(waits for cache miss)	(waits for cache miss)	Response to P2's cache miss	Update memory with block from P0
5		(waits for cache miss)	Tests lock = 0; succeeds	Bus services P1's cache miss	
6		Tests lock = 0; succeeds	Attempt swap; needs write permission	Response to P1's cache miss	Responds to P1's cache miss; sends lock variable
7		Attempt swap; needs write permission	Send invalidate to gain write permission	Bus services P2's invalidate	
8		Cache miss	Swap; reads lock = 0 and sets to 1	Bus services P1's cache miss	
9		Swap; read lock = 1 sets to 1; go back to spin	Responds to P1's cache miss, sends lock = 1	Response to P2's cache miss	

FIGURE 9.3.6 Cache coherence steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write-invalidate coherency. P0 starts with the lock (step 1). P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3–5). P2 wins and enters the *critical section* (steps 6 and 7), while P1 spins and waits (steps 7 and 8). The "critical section" is the name for the code between the lock and the unlock. When P2 exits the critical section, it sets the lock to 0, which will invalidate the copy in P1's cache, restarting the process.

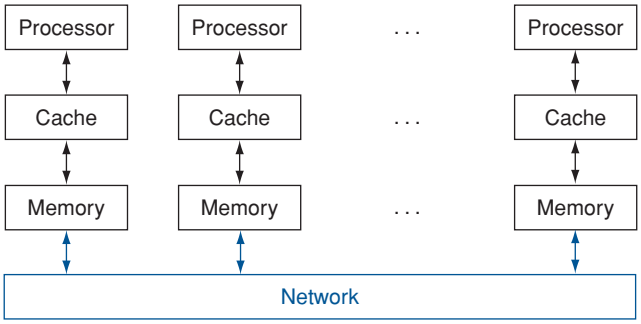


FIGURE 9.4.1 The organization of a network-connected multiprocessor. Note that, in contrast to Figure 9.3.1 on page 9-12, the multiprocessor connection is no longer between memory and the processor.

attached to each processor, and the connection medium—the network—is between these combined nodes. For single-bus systems, the medium is used on every memory access, while in the latter case it is used only for interprocessor communication.

shared memory A memory for a parallel processor with a single address space, implying implicit communication with loads and stores.

distributed memory Physical memory that is divided into modules, with some placed near each processor in a multiprocessor.

cache coherent NUMACC-NUMA A nonuniform memory access multiprocessor that maintains coherence for all caches.

This brings us to an old debate about the organization of memory in large-scale parallel processors. The debate unfortunately often centers on a false dichotomy: **shared memory** versus **distributed memory**. Shared memory really means a single address space, implying implicit communication with loads and stores. The real opposite of a single address is *multiple private memories,* implying explicit communication with sends and receives.

The three general categories of multiprocessors are the following:

- *Uniform memory access* (UMA): Memory access time is the same, no matter where the word is located. This is normal with a common memory.
- *Nonuniform memory access* (NUMA): Memory access time varies depending on its address. This is normal with a physically distributed memory.
- **Cache-coherent nonuniform memory access (CC-NUMA)**: A NUMA with a mechanism that ensures cache coherence.

In machines without a single global address space, communication is explicit; the programmer or the compiler must send messages to ship data to another node and must receive messages to accept data from another node.

EXAMPLE

Parallel Program (Message Passing)

Let's try our summing example again for a network-connected multiprocessor with 100 processors using multiple private memories.

ANSWER

Since this computer has multiple address spaces, the first step is distributing the 100 subsets to each of the local memories. The processor containing the 100,000 numbers sends the subsets to each of the 100 processor-memory nodes.

The next step is to get the sum of each subset. This step is simply a loop that every execution unit follows; read a word from local memory and add it to a local variable:

```
sum = 0;
for (i = 0; i<1000; i = i + 1) /* loop over each array */
 sum = sum + A1[i]; /* sum the local arrays */
```

The last step is adding these 100 partial sums. The hard part is that each partial sum is located in a different execution unit. Hence, we must use the interconnection network to send partial sums to accumulate the final sum. Rather than sending all the partial sums to a single processor, which would result in sequentially adding the partial sums, we again divide to conquer. First, half of the execution units send their partial sums to the other half of the execution units, where two partial sums are added together. Then one quarter of the execution units (half of the half) send this new partial sum to the other quarter of the execution units (the remaining half of the half) for the next round of sums. This halving, sending, and receiving continues until there is a single sum of all numbers. Let `Pn` represent the number of the execution unit, `send(x,y)` be a routine that sends over the interconnection network to execution unit number `x` the value `y`, and `receive()` be a function that accepts a value from the network for this execution unit:

```
limit = 100; half = 100;/* 100 processors */
repeat
  half = (half+1)/2; /* send vs. receive dividing line*/
  if (Pn >= half && Pn < limit) send(Pn - half, sum);
  if (Pn < (limit/2)) sum = sum + receive();
  limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

This code divides all processors into senders or receivers and each receiving processor gets only one message, so we can presume that a receiving processor will stall until it receives a message. Thus, send and receive can be used as primitives for synchronization as well as for communication, as the processors are aware of the transmission of data.

If there is an odd number of nodes, the middle node does not participate in send/receive. The limit is then set so that this node is the highest node in the next iteration.

Addressing in Large-Scale Parallel Processors

Most commercial, large-scale processors use memory that is distributed; otherwise it is either very difficult or very expensive to build a machine that can scale up to scores of processors with scores of memory modules.

The next question facing distributed-memory machines is communication. For the hardware designer, the simplest solution is to offer only send and receive instead of the implicit communication that is possible as part of any load or store. Send and receive also have the advantage of making it easier for the programmer to optimize communication: It's simpler to overlap computation with communication by using explicit sends and receives rather than with implicit loads and stores.

Hardware Software Interface

Adding a software layer to provide a single address space on top of sends and receives so that communication is possible as part of any load or store is harder, although it is comparable to the virtual memory system already found in most processors (see Chapter 7). In virtual memory, a uniprocessor uses page tables to decide if an address points to data in local memory or on a disk; this translation system might be modified to decide if the address points to local data, to data in another processor's memory, or to disk. Although *shared virtual memory*, as it is called, creates the illusion of shared memory—just as virtual memory creates the illusion of a very large memory—since it invokes the operating system, performance is usually so slow that shared-memory communication must be rare or else most of the time is spent transferring pages.

distributed shared memory (DSM) A memory scheme that uses addresses to access remote data when demanded rather than retrieving the data in case it might be used.

On the other hand, loads and stores normally have much lower communication overhead than do sends and receives. Moreover, some applications will have references to remote information that is only occasionally and unpredictably accessed, so it is much more efficient to use an address to remote data when *demanded* rather than to retrieve it in case it *might* be used. Such a machine has **distributed shared memory** (DSM).

Caches are important to performance no matter how communication is performed, so we want to allow the shared data to appear in the cache of the processor that owns the data as well as in the processor that requests the data. Thus, the single global address space in a network-connected multiprocessor resurrects the issue of cache coherency, since there are multiple copies of the same data with the same address in different processors. Clearly, the bus-snooping protocols of Section 9.3 won't work here, as there is no single bus on which all memory references are broadcast. Since the designers of the Cray T3E had no bus to support cache coherency, the T3E has a single address space but it is not cache coherent.

directory A repository for information on the state of every block in main memory, including which caches have copies of the block, whether it is dirty, and so on. Used for cache coherence.

A cache-coherent alternative to bus snooping is **directories**. In directory-based protocols, logically a single directory keeps the state of every block in main memory. Information in the directory can include which caches have copies of the block, whether it is dirty, and so on. Fortunately, directory entries can be distributed so that different requests can go to different memories, thereby reducing contention and allowing a scalable design. Directories retain the characteristic that the sharing status of a block is always in a single known location, making a large-scale parallel processor plausible.

Designers of snooping caches and directories face similar issues; the only difference is the mechanism that detects when there is a write to shared data. Instead of watching the bus to see if there are requests that require that the local cache be updated or invalidated, the directory controller sends explicit commands to each processor that has a copy of the data. Such messages can then be sent over the network.

Note that with a single address space, the data could be placed arbitrarily in memories of different processors. This has two negative performance consequences. The first is that the miss penalty would be much longer because the request must go over the network. The second is that the network bandwidth would be consumed moving data to the proper processors. For programs that have low miss rates, this may not be significant. On the other hand, programs with high miss rates will have much lower performance when data is randomly assigned.

If the programmer or the compiler allocates data to the processor that is likely to use it, then this performance pitfall is removed. Unlike private memory organizations, this allocation only needs to be good, since missing data can still be fetched. Such leniency simplifies the allocation problem.

Hardware Software Interface

Since the number of pins per chip is limited, not all processors can be connected directly to each other. This restriction has inspired a whole zoo of topologies for consideration in the design of the network. In Section 9.6, we'll look at the characteristics of some of the key alternatives of network designs. First, let's look at another way to connect computers by networks.

9.5 Clusters

There are many mainframe applications—such as databases, file servers, Web servers, simulations, and multiprogramming/batch processing—amenable to running on more loosely coupled machines than the cache-coherent NUMA machines of the prior section. These applications often need to be highly available, requiring some form of fault tolerance and repairability. Such applications—plus the similarity of the multiprocessor nodes to desktop computers and the emergence of high-bandwidth, switch-based local area networks—are why large-scale processing uses *clusters* of off-the-shelf, whole computers.

One drawback of clusters has been that the cost of administering a cluster of N machines is about the same as the cost of administering N independent machines, while the cost of administering a shared address space multiprocessor with N processors is about the same as administering a single machine.

Another drawback is that clusters are usually connected using the I/O bus of the computer, whereas multiprocessors are usually connected on the memory bus of the computer. The memory bus has higher bandwidth, allowing multiprocessors to drive the network link at higher speed and to have fewer conflicts with I/O traffic on I/O-intensive applications.

A final weakness is the division of memory: a cluster of N machines has N independent memories and N copies of the operating system, but a shared address multiprocessor allows a single program to use almost all the memory in the computer. Thus, a sequential program in a cluster has 1/Nth the memory available compared to a sequential program in an SMP.

The major distinction between the two is the purchase price for equivalent computing power for large-scale machines. Since large-scale multiprocessors have small volumes, the extra development costs of large machines must be amortized over few systems, resulting in higher cost to the customer. Since the same switches sold in high volume for small systems can be composed to construct large networks for large clusters, local area network switches have the same economy-of-scale advantages as small computers.

The weakness of separate memories for program size turns out to be a strength in system availability. Since a cluster consists of independent computers connected through a local area network, it is much easier to replace a machine without bringing down the system in a cluster than in an SMP. Fundamentally, the shared address means that it is difficult to isolate a processor and replace a processor without heroic work by the operating system. Since the cluster software is a layer that runs on top of local operating systems running on each computer, it is much easier to disconnect and replace a broken machine.

Given that clusters are constructed from whole computers and independent, scalable networks, this isolation also makes it easier to expand the system without bringing down the application that runs on top of the cluster. High availability and rapid, incremental expandability make clusters attractive to service providers for the World Wide Web.

As is often the case with two competing solutions, each side tries to borrow ideas from the other to become more attractive.

On one side of the battle, to combat the high-availability weakness of multiprocessors, hardware designers and operating system developers are trying to offer the ability to run multiple operating systems on portions of the full machine, so that a node can fail or be upgraded without bringing down the whole machine.

On the other side of the battle, since both system administration and memory size limits are approximately linear in the number of independent machines, some are reducing the cluster problems by constructing clusters from small-scale SMPs. For example, a cluster of 32 processors might be constructed from eight four-way SMPs or four eight-way SMPs. Such "hybrid" clusters—sometimes called **constellations** or *clustered, shared memory*—are proving popular with applications that care about cost/performance, availability, and expandability. Figure 9.1.1 on page 9-5 shows that about half of the clusters in the Top 500 supercomputers contain single-processor workstations and about half of the clusters contain SMP servers.

constellation A cluster that uses an SMP as the building block.

9.6 Network Topologies

Chapter 8 reviewed off-the-shelf, switched, local area networks that are the foundation of clusters. In this section we describe proprietary networks used in multiprocessors.

Network costs include the number of switches, the number of links on a switch to connect to the network, the width (number of bits) per link, and length of the links when the network is mapped into a physical machine. For example, on a machine that scales between tens and hundreds of processors, some links may be metal rectangles within a chip that are a few millimeters long, and others may be cables that must stretch several meters from one cabinet to another. Network performance is multifaceted as well. It includes the latency on an unloaded network to send and receive a message, the throughput in terms of the maximum number of messages that can be transmitted in a given time period, delays caused by contention for a portion of the network, and variable performance depending on the pattern of communication. Another obligation of the network may be fault tolerance, since very large systems may be required to operate in the presence of broken components.

Networks are normally drawn as graphs, with each arc of the graph representing a link of the communication network. The processor-memory node is shown as a black square, and the switch is shown as a colored circle. In this section, all links are *bidirectional;* that is, information can flow in either direction. All networks consist of *switches* whose links go to processor-memory nodes and to other switches. The first improvement over a bus is a network that connects a sequence of nodes together:

This topology is called a *ring*. Since some nodes are not directly connected, some messages will have to hop along intermediate nodes until they arrive at the final destination.

Unlike a bus, a ring is capable of many simultaneous transfers. Because there are numerous topologies to choose from, performance metrics are needed to distinguish these designs. Two are popular. The first is *total* **network bandwidth**, which is the bandwidth of each link multiplied by the number of links. This represents the very best case. For the ring network above with P processors, the total network bandwidth would be P times the bandwidth of one link; the total network bandwidth of a bus is just the bandwidth of that bus, or two times the bandwidth of that link.

network bandwidth Informally, the peak transfer rate of a network; can refer to the speed of a single link or the collective transfer rate of all links in the network.

To balance this best case, we include another metric that is closer to the worst case: the *bisection bandwidth*. This is calculated by dividing the machine into two parts, each with half the nodes. Then you sum the bandwidth of the links that cross that imaginary dividing line. The bisection bandwidth of a ring is two times the link bandwidth, and it is one times the link bandwidth for the bus. If a single link is as fast as the bus, the ring is only twice as fast as a bus in the worst case, but it is P times faster in the best case.

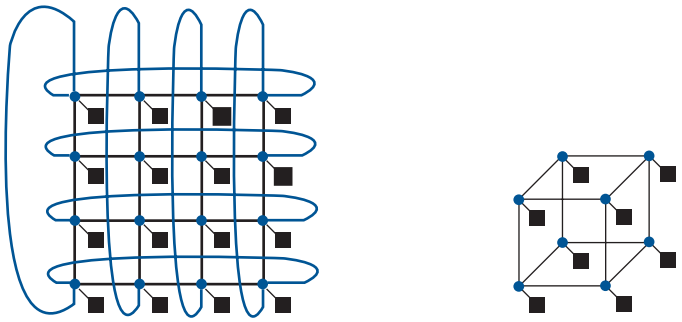
Since some network topologies are not symmetric, the question arises of where to draw the imaginary line when bisecting the machine. This is a worst-case metric, so the answer is to choose the division that yields the most pessimistic network performance; stated alternatively, calculate all possible bisection bandwidths and pick the smallest. We take this pessimistic view because parallel programs are often limited by the weakest link in the communication chain.

fully connected network
A network that connects processor-memory nodes by supplying a dedicated communication link between every node.

At the other extreme from a ring is a **fully connected network**, where every processor has a bidirectional link to every other processor. For fully connected networks, the total network bandwidth is $P \times (P - 1)/2$, and the bisection bandwidth is $(P/2)^2$.

The tremendous improvement in performance of fully connected networks is offset by the tremendous increase in cost. This inspires engineers to invent new topologies that are between the cost of rings and the performance of fully connected networks. The evaluation of success depends in large part on the nature of the communication in the workload of parallel programs run on the machine.

The number of different topologies that have been discussed in publications would be difficult to count, but the number that have been used in commercial parallel processors is just a handful. Figure 9.6.1 illustrates two of the popular



a. 2-D grid or mesh of 16 nodes b. n-cube tree of 8 nodes (8 = 2^3 so n = 3)

FIGURE 9.6.1 Network topologies that have appeared in commercial parallel processors. The colored circles represent switches and the black squares represent processor-memory nodes. Even though a switch has many links, generally only one goes to the processor. The Boolean n-cube topology is an n-dimensional interconnect with 2^n nodes, requiring n links per switch (plus one for the processor) and thus n nearest-neighbor nodes. Frequently these basic topologies have been supplemented with extra arcs to improve performance and reliability.

topologies. Real machines frequently add extra links to these simple topologies to improve performance and reliability.

An alternative to placing a processor at every node in a network is to leave only the switch at some of these nodes. The switches are smaller than processor-memory-switch nodes, and thus may be packed more densely, thereby lessening distance and increasing performance. Such networks are frequently called **multistage networks** to reflect the multiple steps that a message may travel. Types of multistage networks are as numerous as single-stage networks; Figure 9.6.2 illustrates

multistage network
A network that supplies a small switch at each node.

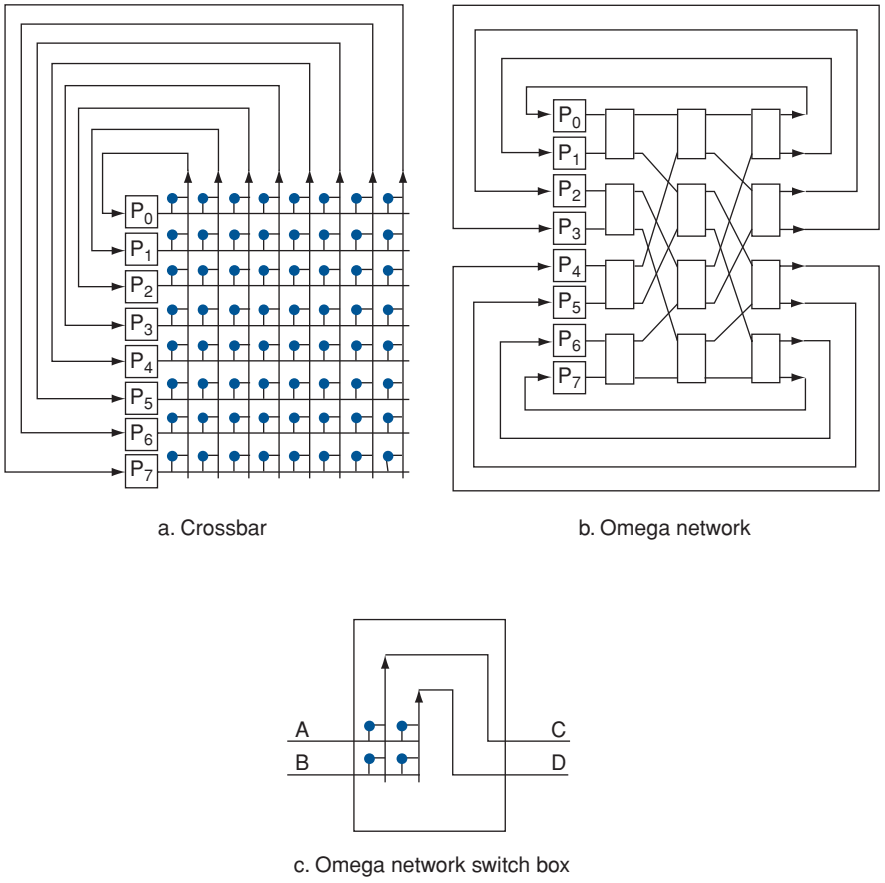


FIGURE 9.6.2 Popular multistage network topologies for eight nodes. The switches in these drawings are simpler than in earlier drawings because the links are unidirectional; data comes in at the bottom and exits out the right link. The switch box in *c* can pass A to C and B to D or B to C and A to D. The crossbar uses n^2 switches, where n is the number of processors, while the Omega network uses $n/2 \log_2 n$ of the large switch boxes, each of which is logically composed of four of the smaller switches. In this case, the crossbar uses 64 switches versus 12 switch boxes, or 48 switches, in the Omega network. The crossbar, however, can support any combination of messages between processors, while the Omega network cannot.

fully connected network A network that connects processor-memory nodes by supplying a dedicated communication link between every node.

crossbar network A network that allows any node to communicate with any other node in one pass through the network.

two of the popular multistage organizations. A **fully connected** or **crossbar network** allows any node to communicate with any other node in one pass through the network. An *Omega network* uses less hardware than the crossbar network ($2n \log_2 n$ versus n^2 switches), but contention can occur between messages, depending on the pattern of communication. For example, the Omega network in Figure 9.6.2 cannot send a message from P0 to P6 at the same time it sends a message from P1 to P7.

Implementing Network Topologies

This simple analysis of all the networks in this section ignores important practical considerations in the construction of a network. The distance of each link affects the cost of communicating at a high clock rate—generally, the longer the distance, the more expensive it is to run at a high clock rate. Shorter distances also make it easier to assign more wires to the link, as the power to drive many wires from a chip is less if the wires are short. Shorter wires are also cheaper than longer wires. A final practical limitation is that the three-dimensional drawings must be mapped onto chips and boards that are essentially two-dimensional media. The bottom line is that topologies that appear elegant when sketched on the blackboard may look awkward when constructed from chips, cables, boards, and boxes.

9.7 Multiprocessors Inside a Chip and Multithreading

An alternative to multiple microprocessors sharing an interconnect is bringing the processors inside the chip. In such designs, the processors typically share some of the caches and the external memory interface. Clearly, the latencies associated with chip-to-chip communication disappear when everything is on the same chip. The question is, What is the impact of the processors on the memory hierarchy? If they are running the same code, such as a database, then the processors at least amortize the instruction accesses. Shared data structures are also much less of a problem when the caches are shared as well. Several companies have announced microprocessors with multiple cores per chip.

The idea of increasing utilization of resources on a chip via parallel execution of threads has found another implementation. *Multithreading* allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogram-

ming. In addition, the hardware must support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles.

There are two main approaches to multithreading. *Fine-grained multithreading* switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time. To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle. One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. The primary disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level 2 cache misses. This change relieves the need to have thread switching be essentially free and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: It is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grained multithreading. Because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen. The new thread that begins executing after the stall must fill the pipeline before instructions will be able to complete. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit thread-level parallelism at the same time it exploits instruction-level parallelism. The key insight that motivates SMT is that modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Figure 9.7.1 conceptually illustrates the differences in a processor's ability to exploit superscalar resources for the following processor configurations. The top portion shows how four threads would execute independently on a superscalar with no multithreading support. The bottom portion shows how the four threads

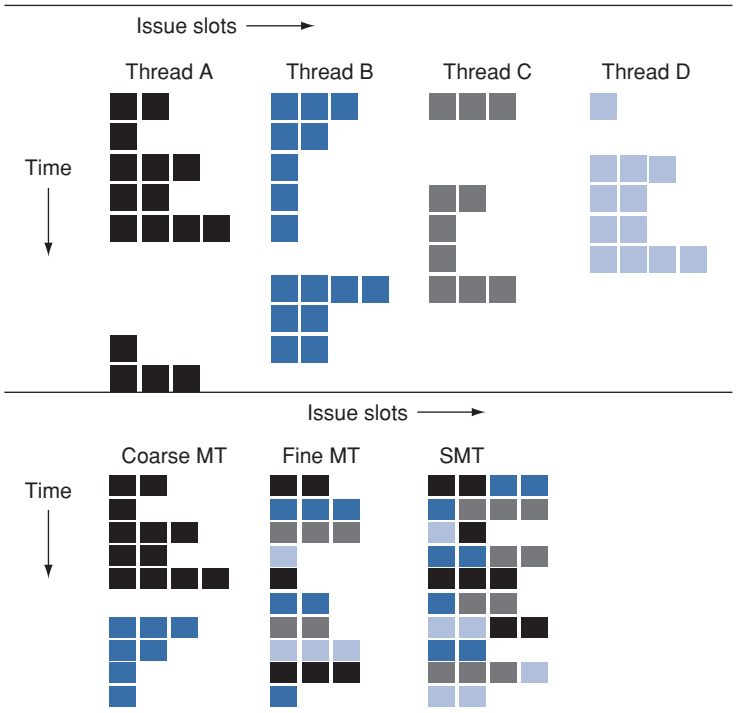


FIGURE 9.7.1 How four threads use the issue slots of a superscalar processor in different approaches. The four threads at the top show how each would execute on a standard superscalar processor without multithreading support. The three examples at the bottom show how they would execute together in three multithreading options. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. The additional pipeline start-up effects for coarse MT, which are not illustrated in this figure, would lead to further loss in throughput for coarse MT.

could be combined to execute on the processor more efficiently using three multithreading options:

- A superscalar with coarse-grained multithreading
- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading

In the superscalar without multithreading support, the use of issue slots is limited by a lack of instruction-level parallelism. In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor.

Although this reduces the number of completely idle clock cycles, within each clock cycle, the instruction-level parallelism limitations still lead to idle cycles. Furthermore, in a coarse-grained multithreaded processor, since thread switching only occurs when there is a stall and the new thread has a start-up period, there are likely to be some fully idle cycles remaining.

In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle, however, instruction-level parallelism limitations still lead to a significant number of idle slots within individual clock cycles.

In the SMT case, thread-level parallelism (TLP) and instruction-level parallelism (ILP) are exploited simultaneously, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors—including how many active threads are considered, finite limitations on buffers, the ability to fetch enough instructions from multiple threads, and practical limitations of what instruction combinations can issue from one thread and from multiple threads—can also restrict how many slots are used. Although Figure 9.7.1 greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

As mentioned earlier, simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the integrated exploitation of TLP through multithreading. In particular, dynamically scheduled superscalar processors have a large set of registers that can be used to hold the register sets of independent threads (assuming separate renaming tables are kept for each thread). Because register renaming provides unique register identifiers, instructions from multiple threads can be mixed in the data path without confusing sources and destinations across the threads. This observation leads to the insight that multithreading can be built on top of an out-of-order processor by adding a per-thread renaming table, keeping separate PCs, and providing the capability for instructions from multiple threads to commit. There are complications in handling instruction commit, since we would like instructions from independent threads to be able to commit independently. The independent commitment of instructions from separate threads can be supported by logically keeping a separate reorder buffer for each thread.

There is a variety of other design challenges for an SMT processor, including the following:

- Dealing with a larger register file needed to hold multiple contexts
- Maintaining low overhead on the clock cycle, particularly in critical steps such as instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging

- Ensuring that the cache conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation

In viewing these problems, two observations are important. First, in many cases, the potential performance overhead due to multithreading is small, and simple choices work well enough. Second, the efficiency of current superscalars is low enough that there is room for significant improvement, even at the cost of some overhead. SMT appears to be the most promising way to achieve that improvement in throughput.

Intel calls its SMT support in the Pentium 4 *Hyper-Threading*. It supports just two threads by doubling the IA-32 architectural state, and they share all the caches and functional units.

9.8 Real Stuff: The Google Cluster of PCs

Search engines have a major reliability requirement, since people are using them at all times of the day and from all over the world. Google must essentially be continuously available.

Since a search engine is normally interacting with a person, its latency must not exceed its users' patience. Google's goal is that no search takes more than 0.5 seconds, including network delays. Bandwidth is also vital. Google serves an *average* of about 1000 queries per second and has searched and indexed more than 3 billion pages.

In addition, a search engine must crawl the Web regularly to have up-to-date information to search. Google crawls the entire Web and updates its index every four weeks, so that every Web page is visited once a month. Google also keeps a local copy of the text of most pages so that it can provide the snippet text as well as offer a cached copy of the page.

To keep up with such demand, Google uses more than 6000 processors and 12,000 disks, giving Google a total of about 1 petabyte of disk storage.

Rather than achieving availability by using RAID storage, Google relies on redundant sites, each with thousands of disks and processors: two sites in Silicon Valley and two in Virginia. The search index, which is a small number of terabytes, plus the repository of cached pages, which is on the order of the same size, are replicated across the sites. Thus, if a single site fails, there are still two more sites that can sustain the service. In addition, the index and repository are replicated within a site to help share the workload as well as to continue to provide service within a site even if components fail.

Each site is connected to the Internet via OC48 (2488 Mbit/sec) links of the collocation site. To provide against failure of the collocation link, there is a sepa-

rate OC12 link connecting a pair of sites so that in an emergency both sites can use the Internet link at one site. The external link is unlikely to fail at both sites since different network providers supply the OC48 lines.

Figure 9.8.1 shows the floor plan of a typical site. The OC48 link connects to two 128 × 128 Ethernet switches via a large switch. Note that this link is also connected to the rest of the servers in the site. These two switches are redundant so that a switch failure does not disconnect the site. There is also an OC12 link from the 128 × 128 Ethernet switches to the sister site for emergencies. Each switch can connect to 128 1-Gbit/sec Ethernet lines. Racks of PCs, each with 4 1-Gbit/sec Ethernet interfaces, are connected to the two 128 × 128 Ethernet switches. Thus, a single site can support 2 × 128/4 or 64 racks of PCs.

Figure 9.8.2 shows Google's rack of PCs. Google uses PCs that are only 1 VME rack unit. To connect these PCs to the 128 × 128 Ethernet switches, it uses a small Ethernet switch. It is 4 RU high, leaving room in the rack for 40 PCs. This switch has modular network interfaces, which are organized as removable *blades*. Each

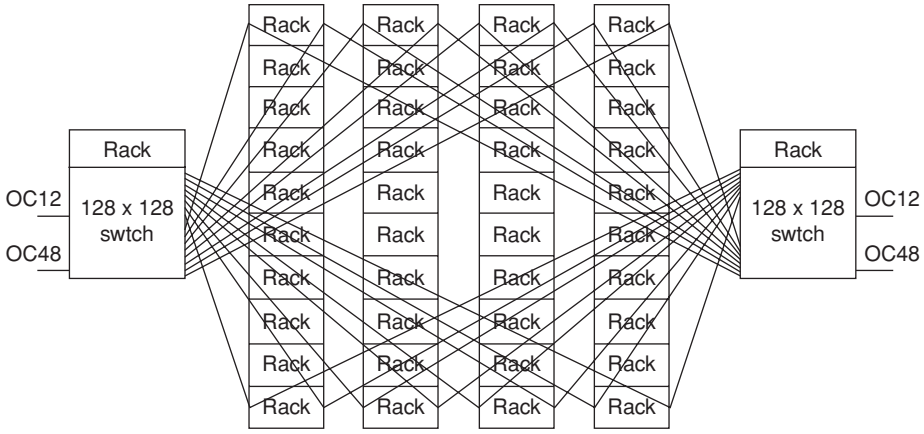


FIGURE 9.8.1 Floor plan of a Google cluster, from a God's-eye view. There are 40 racks, each connected via 4 copper Gbit Ethernet links to two redundant 128 × 128 switches. Figure 9.8.2 shows a rack contains 80 PCs, so this facility has about 3200 PCs. (For clarity, the links are only shown for the top and bottom rack in each row.) These racks are on a raised floor so that the cables can be hidden and protected. Each 128 × 128 Ethernet switch in turn is connected to the collocation site network via an OC48 (2.4 Gbit) link to the Internet. There are two 128 × 128 Ethernet switches so that the cluster is still connected even if one switch fails. There is also a separate OC12 (622 Mbit) link to a separate nearby collocation site in case the OC48 network of one collocation site fails; it can still serve traffic over the OC12 to the other site's network. Each 128 × 128 Ethernet switch can handle 128 1-Gbit Ethernet lines, and each rack has 2 1-Gbit Ethernet lines per switch, so the maximum number of racks for the site is 64. The two racks near the 128 × 128 Ethernet switches contain a few PCs to act as front ends and help with tasks such as html service, load balancing, monitoring, and UPS to keep the switch and fronts up in case of a short power failure. It would seem that a facility that has redundant diesel engines to provide independent power for the whole site would make UPS redundant. A survey of data center users suggests power failures still happen yearly.

blade can contain 8 100-Mbit/sec Ethernet interfaces or a single 1-Gbit/sec Ether-

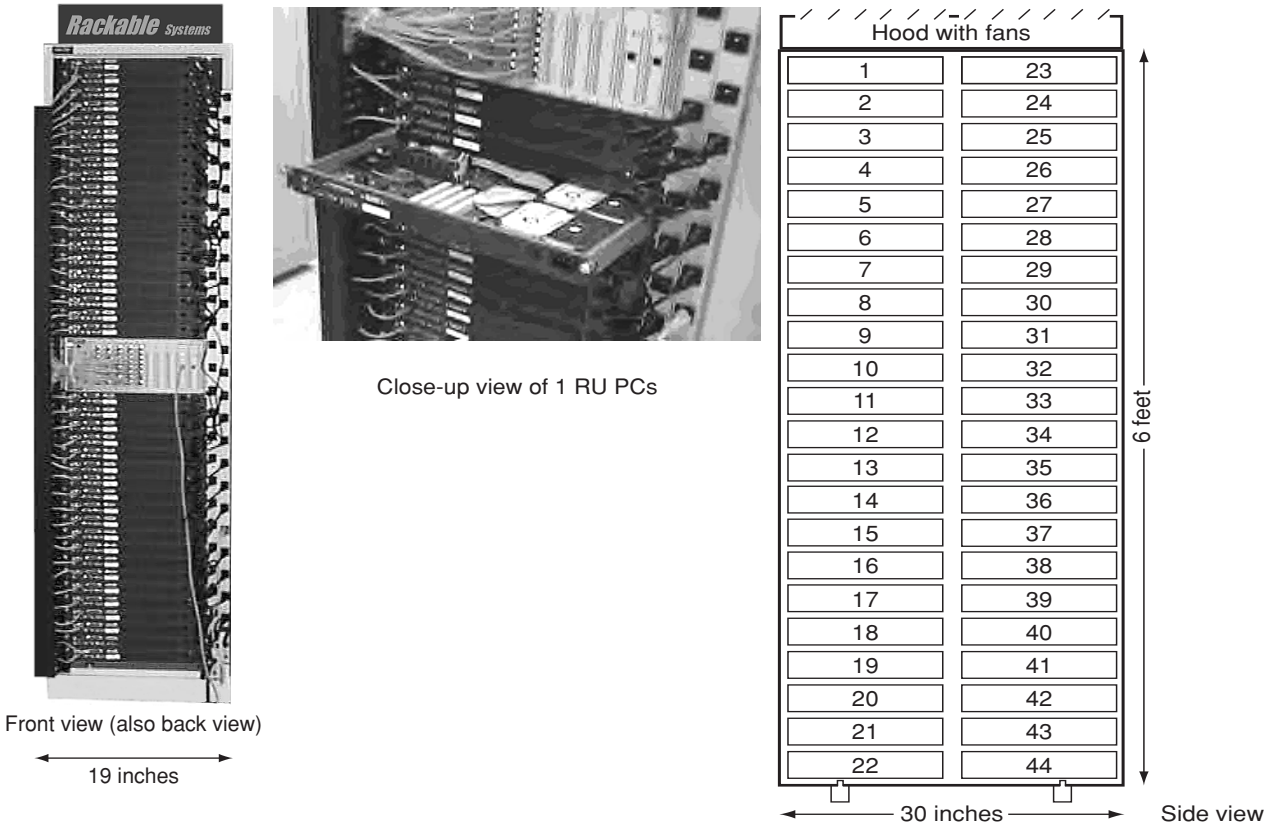


FIGURE 9.8.2 Front view, side view, and close-up of a rack of PCs used by Google. The photograph on the left shows the small Ethernet switch in the middle, with 20 PCs above and 20 PCs below in the Rackable Systems rack. Each PC connects via a Cat5 cable on the left side to the switch in the middle, running 100-Mbit Ethernet. Each "blade" of the switch can hold eight 100-Mbit Ethernet interfaces or one 1-Gbit interface. There are also two 1-Gbit Ethernet links leaving the switch on the right. Thus, each PC has only two cables: one Ethernet and one power cord. The far right of the photo shows a power strip, with each of the 40 PCs and the switch connected to it. Each PC is 1 VME rack unit (RU) high. The switch in the middle is 4 RU high. The photo in the middle is a close-up of a rack, showing contents of a 1 RU PC. This unit contains two 5400 RPM IDE drives on the right of the box, 256 MB of 100 MHz SDRAM, a PC motherboard, a single power supply, and an Intel microprocessor. Each PC runs versions 2.2.16 or 2.2.17 Linux kernels on a slightly modified Red Hat release. You can see the Ethernet cables on the left, power cords on the right, and table Ethernet cables connected to the switch at the top of the figure. In December 2000 the unassembled parts costs were about $500 for the two drives, $200 for the microprocessor, $100 for the motherboard, and $100 for the DRAM. Including the enclosure, power supply, fans, cabling, and so on, an assembled PC might cost $1300 to $1700. The drawing on the right shows that PCs are kept in two columns, front and back, so that a single rack holds 80 PCs and two switches. The typical power per PC is about 55 watts and about 70 watts per switch, so a rack uses about 4500 watts. Heat is exhausted into a 3-inch vent between the two columns, and the hot air is drawn out the top using fans. (The drawing shows 22 PCs per side, each 2 RU high, instead of the Google configuration of 40 1 RU PCs plus a switch per side.)

net interface. Thus, 5 blades are used to connect 100-Mbit/sec Cat5 cables to each of the 40 PCs in the rack, and 2 blades are used to connect 1-Gbit/sec copper cables to the two 128 × 128 Ethernet switches.

To pack even more PCs, Google selected a rack that offers the same configuration in the *front and back*, yielding 80 PCs and two switches per rack, as Figure 9.8.2 shows. This system has about a 3-inch gap in the middle between the columns of PCs for the hot air to exit, which is drawn out of the "chimney" via exhaust fans at the top of the rack.

The PC itself was fairly standard: two ATA/IDE drives, 256 MB of SDRAM, a modest Intel microprocessor, a PC motherboard, one power supply, and a few fans. Each PC runs the Linux operating system. To get the best value per dollar, every 2–3 months Google increases the capacity of the drives or the speed of the processor. Thus, the 40-rack site shown in 9.8.1 was populated with microprocessors that varied from a 533 MHz Celeron to an 800 MHz Pentium III, disks that varied in capacity between 40 and 80 GB and in speed between 5400 and 7200 RPM, and memory bus speed that was either 100 or 133 MHz.

Performance

Each collocation site connects to the Internet via OC48 (2488 Mbit/sec) links, which is shared by Google and the other Internet service providers. If a typical response to a query is, say, 4000 bytes, and if Google serves 100 million queries per day, then the average bandwidth demand is

$$\frac{100{,}000{,}000 \text{ queries/day} \times 4000 \text{ bytes/query} \times 8 \text{ bits/byte}}{24 \times 60 \times 60 \text{ sec/day}} = \frac{3{,}200{,}000 \text{ Mbits}}{86{,}400 \text{ sec}} \approx 37 \text{ Mbits/sec}$$

which is just 1.5% of the link speed of each site. Even if we multiply by a factor of 4 to account for peak versus average demand and requests as well as responses, Google needs little of that bandwidth.

Crawling the Web and updating the sites needs much more bandwidth than serving the queries. Let's estimate some parameters to put things into perspective. Assume that it takes 7 days to crawl 3 billion pages:

$$\frac{3{,}000{,}000{,}000 \text{ pages} \times 4000 \text{ bytes/page} \times 8 \text{ bits/byte}}{24 \times 60 \times 60 \text{ sec/day} \times 7 \text{ days}} = \frac{96{,}000{,}000 \text{ Mbits}}{604{,}800 \text{ sec}} \approx 159 \text{ Mbits/sec}$$

These data are collected at a single site, but the final multiterabyte index and repository must then be replicated at the other two sites. If we assume we have 7 days to replicate the data and that we are shipping, say, 15 TB from one site to two sites, then the average bandwidth demand is

$$2 \times \frac{15{,}000{,}000 \text{ MB} \times 8 \text{ bits/byte}}{24 \times 60 \times 60 \text{ sec/day} \times 7 \text{days}} = \frac{240{,}000{,}000 \text{ Mbits}}{604{,}800 \text{ sec}} \approx 396 \text{ Mbits/sec}$$

Hence, the machine-to-person bandwidth is relatively trivial, with the real bandwidth demand being machine to machine. This is still a small fraction of the 2488 Mbits/sec available.

Time of flight for messages across the United States takes about 0.1 seconds, so it's important for Europe to be served from the Virginia sites and for California to be served by Silicon Valley sites. To try to achieve the goal of 0.5 second latency, Google software normally guesses where the search is from in order to reduce time-of-flight delays.

Cost

Given that the basic building block of the Google cluster is a PC, the capital cost of a site is typically a function of the cost of a PC. Rather than buy the latest microprocessor, Google looks for the best cost-performance. We estimate the PC cost was $1300 to $1700.

The switches cost about $1500 for the small Ethernet switch and about $100,000 each for the 128 × 128 Ethernet switches. If the racks themselves cost about $1000 to $2000 each, the total capital cost of a 40-rack site is about $4.5 million to $6.0 million. Including 3200 microprocessors and 0.8 TB of DRAM, the disk storage costs about $10,000 to $15,000 per terabyte. Had they purchased standard servers and disk arrays in that time frame, their cost would have been 5 to 10 times higher.

The Google rack with 80 PCs, with each PC operating at about 55 W, uses 4500 W in 10 square feet. It is considerably higher than the 1000 W per rack expected by the collocation sites. Each Google rack also uses 60 amps. As mentioned above, reducing power per PC is a major opportunity for the future of such clusters, especially as the cost per kilowatt-hour is increasing and the cost per Mbits/sec is decreasing.

Reliability

The biggest source of component failures in the Google PC is software. On an average day, about 20 machines will be rebooted, and that normally solves the problem. To reduce the number of cables per PC as well as cost, Google has no ability to remotely reboot a machine. The software stops giving work to a machine when it observes unusual behavior, the operator calls the collocation site and tells them the location of the machine that needs to be rebooted, and a person at the site finds the label and pushes the switch on the front panel. Occasionally the person hits the wrong switch either by mistake or due to mislabeling on the outside of the box.

The next component reliability problem is the hardware, which has about 1/10th the failures of software. Typically, about 2% to 3% of the PCs need to be replaced per year, with failures due to disks and DRAM accounting for 95% of these failures. The remaining 5% are due to problems with the motherboard,

power supply, connectors, and so on. The microprocessors themselves never seem to fail.

The DRAM failures are perhaps a third of the hardware component failures. Google sees errors both from bits changing inside DRAM and when bits transfer over the 100–133 MHz bus. There was no ECC protection available on PC desktop motherboard chip sets at the time, so it was not used. The DRAM is determined to be the problem when Linux cannot be installed with a proper checksum until the DRAM is replaced. Google plans to use ECC both to correct some failures but, more importantly, to make it easier to see when DRAMs fail. The extra cost of the ECC is trivial given the wide fluctuation in DRAM prices; careful purchasing procedures are more important than whether the DIMM has ECC.

Disks are the remaining PC failures. In addition to the standard failures that result in a message to the error log in the console, in almost equal numbers these disks will occasionally result in a *performance failure,* with no error message to the log. Instead of delivering normal read bandwidths at 28 MB/sec, disks will suddenly drop to 4 MB/sec or even 0.8 MB/sec. As the disks are under warranty for five years, Google sends the disks back to the manufacturer for either operational or performance failures to get replacements. Thus, there has been no exploration of the reason for the disk anomalies.

When a PC has problems, it is reconfigured out of the system, and about once a week a person removes the broken PCs. They are usually repaired and then reinserted into the rack.

In regards to the switches, over a two-year period perhaps 200 of the small Ethernet switches were deployed, and 2 or 3 have failed. None of the six 128×128 Ethernet switches has failed in the field, although some have had problems on delivery. These switches have a blade-based design with 16 blades per switch, and 2 or 3 of the blades have failed.

The final issue is collocation reliability. Many Internet service providers experience one power outage a year that affects either the whole site or a major fraction of a site. On average, there is also a network outage so that the whole site is disconnected from the Internet. These outages can last for hours.

Google accommodates collocation unreliability by having multiple sites with different network providers, plus leased lines between pairs of site for emergencies. Power failures, network outages, and so on do not affect the availability of the Google service. Google has not had an outage since the company was a few months old.

Number 9: Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar.

David H. Bailey, "Twelve ways to fool the masses when giving performance results on parallel supercomputers," *Supercomputing Review,* 1991

9.9 Fallacies and Pitfalls

The many assaults on parallel processing have uncovered numerous fallacies and pitfalls. We cover three here.

Pitfall: Measuring performance of parallel processors by linear speedup versus execution time.

"Mortar shot" graphs—plotting performance compared to the number of processors, showing linear speedup, a plateau, and then a falling off—have long been used to judge the success of parallel processors. Although scalability is one facet of a parallel program, it is an indirect measure of performance. The primary question to be asked concerns the power of the processors being scaled: a program that linearly improves performance to equal 100 Intel 80386s may be slower than the sequential version on a single Pentium 4 desktop computer.

Measuring results using linear speedup compared to the execution time can mislead the programmer as well as those hearing the performance claims of the programmer. Many programs with poor speedup are faster than programs that show excellent speedup as the number of processors increases.

Comparing execution times is fair only if you are comparing the best algorithms on each machine. (Of course, you can't subtract time for idle processors when evaluating a parallel processor, so CPU time is an inappropriate metric for parallel processors.) Comparing the identical code on two machines may seem fair, but it is not; the parallel program may be slower on a uniprocessor than a sequential version. Sometimes, developing a parallel program will lead to algorithmic improvements, so that comparing the previously best-known sequential program with the parallel code—which seems fair—compares inappropriate algorithms. To reflect this issue, sometimes the terms *relative speedup* (same program) and *true speedup* (best programs) are used.

Fallacy: Amdahl's law doesn't apply to parallel computers.

In 1987, the head of a research organization claimed that Amdahl's law had been broken by a multiprocessor machine. To try to understand the basis of the media reports, let's see the quote that gave us Amdahl's law [1967, p. 483]:

> *A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.*

This statement must still be true; the neglected portion of the program must limit performance. One interpretation of the law leads to the following lemma: portions of every program must be sequential, so there must be an economic upper bound to the number of processors—say, 100. By showing linear speedup with 1000 processors, this lemma is disproved and hence the claim that Amdahl's law was broken.

The approach of the researchers was to change the input to the benchmark: rather than going 1000 times faster, they computed 1000 times more work in comparable time. For their algorithm, the sequential portion of the pro-

gram was constant, independent of the size of the input, and the rest was fully parallel—hence, linear speedup with 1000 processors. Simply scaling the size of applications, without also scaling floating-point accuracy, the number of iterations, the I/O requirements, and the way applications deal with error may be naive. Many applications will not calculate the correct result if the problem size is increased unwittingly.

We see no reason why Amdahl's law doesn't apply to parallel processors. What this research does point out is the importance of having benchmarks that can grow large enough to demonstrate performance of large-scale parallel processors.

Fallacy: Peak performance tracks observed performance.

One definition of peak performance is "performance that a machine is guaranteed not to exceed." Alas, the supercomputer industry has used this metric in marketing, and its fallacy is being exacerbated with parallel machines. Not only are industry marketers using the nearly unattainable peak performance of a uniprocessor node (see Figure 9.9.1), but also they are then multiplying it by the total number of processors, assuming perfect speedup! Amdahl's law suggests how difficult it is to reach either peak; multiplying the two together also multiplies the sins. Figure 9.9.2 compares the peak to sustained performance on a benchmark; the 64-processor IBM SP2 achieves only 7% of peak performance. Clearly, peak performance does not always track observed performance.

Such performance claims can confuse the manufacturer as well as the user of the machine. The danger is that the manufacturer will develop software libraries

Machine	Peak MFLOPS rating	Harmonic mean MFLOPS of the Perfect Club benchmarks	Percent of peak MFLOPS
Cray X-MP/416	940	14.8	1%
IBM 3090-600S	800	8.3	1%
NEC SX/2	1300	16.6	1%

FIGURE 9.9.1 Peak performance and harmonic mean of actual performance for the 12 Perfect Club benchmarks. These results are for the programs run unmodified. When tuned by hand, performance of the three machines moves to 24.4, 11.3, and 18.3 MFLOPS, respectively. This is still 2% or less of peak performance.

	Cray YMP (8 processors)		IBM SP2 (64 processors)	
	MFLOPS	% Peak	MFLOPS	% Peak
Peak	2,666	100%	14,636	100%
3D FFT PDE	1,795	67%	1,093	7%

FIGURE 9.9.2 Peak versus observed performance for Cray YMP and IBM RS/6000 SP2.

with success judged as percentage of peak performance measured in megaflops rather than taking less time, or that hardware will be added that increases peak node performance but is difficult to use.

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. . . . Demonstration is made of the continued validity of the single processor approach . . .

Gene Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," Spring Joint Computer Conference, 1967

9.10 Concluding Remarks

The dream of building computers by simply aggregating processors has been around since the earliest days of computing. Progress in building and using effective and efficient parallel processors, however, has been slow. This rate of progress has been limited by difficult software problems as well as by a long process of evolving architecture of multiprocessors to enhance usability and improve efficiency. We have discussed many of the software challenges in this chapter, including the difficulty of writing programs that obtain good speedup due to Amdahl's law. The wide variety of different architectural approaches and the limited success and short life of many of the architectures to date has compounded the software difficulties. We discuss the history of the development of these multiprocessors in section 9.11.

Despite this long and checkered past, progress in the last 20 years leads to reasons to be optimistic about the future of parallel processing and multiprocessors. This optimism is based on a number of observations about this progress and the long-term technology directions:

- It is now widely held that the most effective way to build a computer that offers more performance than that achieved with a single-chip microprocessor is by building a multiprocessor or a cluster that leverages the significant price-performance advantages of mass-produced microprocessors.
- Multiprocessors and clusters are highly effective for multiprogrammed workloads, which are often the dominant use of mainframes and large servers, as well as for file servers or Web servers, which are effectively a restricted type of parallel workload. When a workload wants to share resources, such as file storage, or can efficiently time-share a resource, such as a large memory, a multiprocessor can be a very efficient host. Furthermore, the OS software needed to execute multiprogrammed workloads is commonplace.
- The use of parallel processing in domains such as scientific and engineering computation is popular. This application domain has an almost limitless thirst for more computation. It also has many applications that have lots of natural parallelism. Nonetheless, it has not been easy: Programming parallel processors even for these applications remains challenging. Another important application area, which has a much larger market, is large-scale database and transaction-processing systems. This application domain also has

extensive natural parallelism available through parallel processing of independent requests, but its needs for large-scale computation, as opposed to merely access to large-scale storage systems, are less well understood.

- On-chip multiprocessing appears to be growing in importance for two reasons. First, in the embedded market where natural parallelism often exists, such approaches are an obvious alternative to faster, and possibly less efficient, processors. Second, diminishing returns in high-end microprocessor design will encourage designers to pursue on-chip multiprocessing as a potentially more cost-effective direction.

The Future of MPP Architecture

Hennessy and Patterson should move MPPs to Chapter 11.

Jim Gray,
when asked about coverage of MPPs in the second edition of this book, alluding to Chapter 11 bankruptcy protection in U.S. law (1995)

Small-scale multiprocessors built using snooping bus schemes are extremely cost-effective. Microprocessors traditionally have even included much of the logic for cache coherence in the processor chip, and several allow the buses of two or more processors to be directly connected—implementing a coherent bus with no additional logic. With modern integration levels, multiple processors can be placed within a single die, resulting in a highly cost-effective multiprocessor. Recent microprocessors have been including support for NUMA approaches, making it possible to connect small to moderate numbers of processors with little overhead.

What is unclear at present is how the very largest parallel processors will be constructed. The difficulties that designers face include the relatively small market for very large multiprocessors and the need for multiprocessors that scale to larger processor counts to be extremely cost-effective at the lower processor counts, where most of the multiprocessors will be sold. There appear to be four slightly different alternatives for large-scale multiprocessors:

1. Designing a cluster using *all* off-the-shelf components, which offers the lowest cost. The leverage in this approach lies in the use of commodity technology everywhere: in the processors (PC or workstation nodes), in the interconnect (high-speed local area network technology, such as Gigabit Ethernet), and in the software (standard operating systems and programming languages). Of course, such multiprocessors will use message passing, and communication is likely to have higher latency and lower bandwidth than in the alternative designs. For applications that do not need high-bandwidth or low-latency communication, this approach can be extremely cost-effective. Web servers, such as the Google cluster, are a good match.
2. Designing clustered computers that use off-the-shelf processor nodes and a custom interconnect. The advantage of such a design is the cost-effectiveness of the standard processor node, which is often a repackaged desktop computer; the disadvantage is that the programming model will probably need to be message passing even at very small node counts. The

cost of the custom interconnect can be significant and thus make the multiprocessor costly, especially at small node counts. An example is the IBM SP.

3. Large-scale multiprocessors constructed from clusters of midrange multiprocessors with combinations of proprietary and standard technologies to interconnect such multiprocessors. This cluster approach gets its cost-effectiveness using cost-optimized building blocks. Many companies offer a high-end version of such a machine, including HP, IBM, and Sun. Due to the two-level nature of the design, the programming model sometimes must be changed from shared memory to message passing or to a different variation on shared memory, among clusters. This class of machines has made important inroads, especially in commercial applications.
4. Large-scale multiprocessors that simply scale up naturally, using proprietary interconnect and communications controller technology. There are two primary difficulties with such designs. First, the multiprocessors are not cost-effective at small scales, where the cost of scalability is not valued. Second, these multiprocessors have programming models that are incompatible, in varying degrees, with the mainstream of smaller and midrange multiprocessors. The SGI Origin is one example.

Each of these approaches has advantages and disadvantages, and the importance of the shortcomings of any one approach is dependent on the application class. It is unclear which will win out for larger-scale multiprocessors, although the growth of the market for Web servers has made "racks of PCs" the dominant form at least by number of systems.

The Future of Microprocessor Architecture

As we saw in Chapter 6, architects are using ever more complex techniques to try to exploit more instruction-level parallelism. The prospects for finding ever-increasing amounts of instruction-level parallelism in a manner that is efficient to exploit are somewhat limited. As we saw in Chapter 7, there are increasingly difficult problems to be overcome in building memory hierarchies for high-performance processors. Of course, continued technology improvements will allow us to continue to advance clock rate. However, the use of technology improvements that allow a faster gate speed alone is not sufficient to maintain the incredible growth of performance that the industry has experienced for over 20 years. Moreover, as power increases over 100 watts per chip, it is unclear how much higher it can go in air-cooled systems. Hence, power may prove to be another limit to performance.

Unfortunately, for more than a decade, increases in performance have come at the cost of ever-increasing inefficiencies in the use of silicon area, external connections, and power. This diminishing-returns phenomenon has only recently appeared to have slowed the rate of performance growth. What is clear is that we

cannot sustain the rapid rate of performance improvements without significant innovations in computer architecture.

With this in mind, in 2004 it appears that the long-term direction will be to use increased silicon to build multiple processors on a single chip. Such a direction is appealing from the architecture viewpoint—it offers a way to scale performance without increasing hardware complexity. It also offers an approach to easing some of the challenges in memory system design, since a distributed memory can be used to scale bandwidth while maintaining low latency for local accesses. Finally, redundant processors can help with dependability. The challenge lies in software and in what architecture innovations may be used to make the software easier.

In 2000, IBM announced the first commercial chips with two general-purpose processors on a single die, the Power4 processor. Each Power4 contains two processors, a shared secondary cache, an interface to an off-chip tertiary cache or main memory, and a chip-to-chip communication system, which allows a four-processor crossbar-connected module to be built with no additional logic. Using four Power4 chips and the appropriate DRAMs, an eight-processor system can be integrated onto a board about 8 inches on a side. In 2002, Sun announced a chip with four processor cores on a chip, each multithreaded eight ways, presenting the programmer with the illusion of 32 processors. Systems using this chip should ship in 2005. In 2004, Intel announced dual processor chips.

If the number of processors per chip grows with Moore's law, dozens of processors are plausible in the near future. The challenge for such "micromultiprocessors" is the software base that can exploit them, which may lead to opportunities for innovation in program representation and optimization.

Evolution versus Revolution and the Challenges to Paradigm Shifts in the Computer Industry

Figure 9.10.1 shows what we mean by the *evolution-revolution spectrum* of computer architecture innovation. To the left are ideas that are invisible to the user (presumably excepting better cost, better performance, or both) and are at the evolutionary end of the spectrum. At the other end are revolutionary architecture ideas. These are the ideas that require new applications from programmers who must learn new programming languages and models of computation, and must invent new data structures and algorithms.

Revolutionary ideas are easier to get excited about than evolutionary ideas, but to be adopted they must have a much higher payoff. Caches are an example of an evolutionary improvement. Within five years after the first publication about caches, almost every computer company was designing a computer with a cache. The RISC ideas were nearer to the middle of the spectrum, for it took more than eight years for most companies to have a RISC product. Most multiprocessors have tended to the revolutionary end of the spectrum, with the largest-scale multiprocessors (MPPs) being more revolutionary than others.

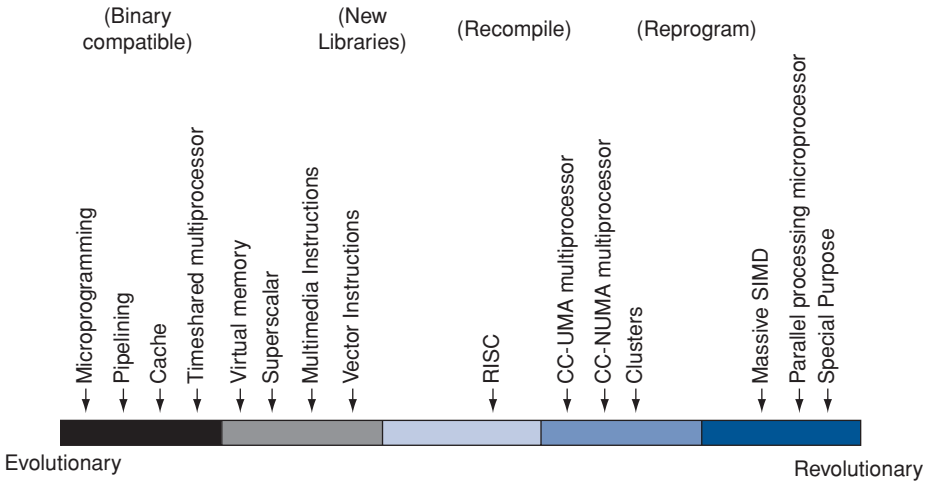


FIGURE 9.10.1 The evolution-revolution spectrum of computer architecture. The first four columns are distinguished from the last column in that applications and operating systems may be ported from other computers rather than written from scratch. For example, RISC is listed in the middle of the spectrum because user compatibility is only at the level of high-level languages (HLLs), while microprogramming allows binary compatibility, and parallel processing multiprocessors require changes to algorithms and extending HLLs. You see several flavors of multiprocessors on this figure. "Timeshared multiprocessor" means multiprocessors justified by running many independent programs at once. "CC-UMA" and "CC-NUMA" mean cache-coherent UMA and NUMA multiprocessors running parallel subsystems such as databases or file servers. Moreover, the same applications are intended for "message passing." "Parallel processing multiprocessor" means a multiprocessor of some flavor sold to accelerate individual programs developed by users. (See section 9.11 to learn about SIMD.)

The challenge for both hardware and software designers who would propose that multiprocessors and parallel processing become the norm, rather than the exception, is the disruption to the established base of programs. There are two possible ways this paradigm shift could be facilitated: if parallel processing offers the only alternative to enhance performance, and if advances in hardware and software technology can construct a gentle ramp that allows the movement to parallel processing, at least with small numbers of processors, to be more evolutionary. Perhaps cost/performance will be replaced with new goals of dependability, security, and/or reduced cost of ownership as the primary justification of such a change.

When contemplating the future—and when inventing your own contributions to the field—remember the hardware/software interface. Acceptance of hardware ideas requires acceptance by software people; therefore, hardware people must learn more about software. In addition, if software people want good machines, they must learn more about hardware to be able to communicate with and thereby influence hardware designers. Also, keep in mind the principles of com-

puter organization found in this book; these will surely guide computers of the future, just as they have guided computers of the past.

9.11 Historical Perspective and Further Reading

As parallelism can appear at many levels, it is useful to categorize the alternatives. In 1966, Flynn proposed a simple model of categorizing computers that is widely used. Scrutinizing the most constrained component of the machine, he counted the number of parallel instruction and data streams and then labeled the computer with this count:

1. *Single instruction stream, single data stream* (SISD, the uniprocessor)
2. *Single instruction stream, multiple data streams* (SIMD)
3. *Multiple instruction streams, single data stream* (MISD)
4. *Multiple instruction streams, multiple data streams* (MIMD)

Some machines are hybrids of these categories, of course, but this classic model has survived because it is simple, easy to understand, and gives a good first approximation. It is also—perhaps because of its understandability—the most widely used scheme.

Your first question about the model should be, "Single or multiple compared with what?" A machine that adds a 32-bit number in 1 clock cycle would seem to have multiple data streams when compared with a bit-serial computer that takes 32 clock cycles to add. Flynn chose computers popular during that time, the IBM 704 and IBM 7090, as the model of SISD; today, the MIPS implementations in Chapters 5 and 6 would be fine reference points.

Single Instruction Multiple Data Computers

SIMD computers operate on vectors of data. For example, when a single SIMD instruction adds 64 numbers, the SIMD hardware sends 64 data streams to 64 ALUs to form 64 sums within a single clock cycle.

The virtues of SIMD are that all the parallel execution units are synchronized and they all respond to a single instruction that emanates from a single program counter (PC). From a programmer's perspective, this is close to the already familiar SISD. Although every unit will be executing the same instruction, each execution unit has its own address registers, and so each unit can have different data addresses.

The original motivation behind SIMD was to amortize the cost of the control unit over dozens of execution units. Another advantage is the reduced size of program memory—SIMD needs only one copy of the code that is being simultaneously executed, while MIMD may need a copy in every processor. Virtual memory and increasing capacity of DRAM chips have reduced the importance of this advantage.

Real SIMD computers have a mixture of SISD and SIMD instructions. There is typically an SISD host computer to perform sequential operations such as branches or address calculations. The SIMD instructions are broadcast to all the execution units, each with its own set of registers and memory. Execution units rely on interconnection networks to exchange data.

data parallelism Parallelism achieved by having massive data.

SIMD works best when dealing with arrays in *for* loops. Hence, for massive parallelism to work in SIMD, there must be massive data, or **data parallelism**. SIMD is at its weakest in *case* or *switch* statements, where each execution unit must perform a different operation on its data, depending on what data it has. Execution units with the wrong data are disabled so that units with proper data may continue. Such situations essentially run at 1/*n*th performance, where *n* is the number of cases.

A basic trade-off in SIMD machines is processor performance versus number of processors. The Connection Machine 2 (CM-2), for example, offers 65,536 single-bit-wide processors, while the Illiac IV had 64 64-bit processors. Figure 9.11.1 lists the characteristics of some well-known SIMD computers.

Surely, the Illiac IV (seen in Figure 9.11.2) is the most infamous of the supercomputer projects. Although successful in pushing several technologies useful in later projects, the Illiac IV failed as a computer. Costs escalated from the $8 million estimated in 1966 to $31 million by 1972, despite the construction of only a quarter of the planned machine. Actual performance was at best 15 MFLOPS compared to initial predictions of 1000 MFLOPS for the full system (see Falk [1976]). Delivered to NASA's Ames Research in 1972, the computer took three more years of engineering before it was operational. For better or worse, com-

Institution	Name	Maximum no. of proc.	Bits/ proc.	Proc. clock rate (MHz)	Number of FPUs	Maximum memory size/system (MB)	Communi-cations BW/system (MB/sec)	Year
U. Illinois	Illiac IV	64	64	13	64	1	2,560	1972
ICL	DAP	4,096	1	5	0	2	2,560	1980
Goodyear	MPP	16,384	1	10	0	2	20,480	1982
Thinking Machines	CM-2	65,536	1	7	2048 (optional)	512	16,384	1987
Maspar	MP-1216	16,384	4	25	0	256 or 1024	23,000	1989

FIGURE 9.11.1 Characteristics of five SIMD computers. Number of FPUs means number of floating-point units.

FIGURE 9.11.2 The Illiac IV control unit followed by its 64 processing elements. It was perhaps the most infamous of supercomputers. The project started in 1965 and ran its first real application in 1976. The 64 processors used a 13-MHz clock, and their combined main memory size was 1 MB: 64×16 KB. The Illiac IV was the first machine to teach us that software for parallel machines dominates hardware issues. Photo courtesy of NASA Ames Research Center.

puter architects are not easily discouraged; SIMD successors of the Illiac IV include the ICL DAP, Goodyear MPP (Figure 9.11.3), Thinking Machines CM-1 and CM-2, and Maspar MP-1 and MP-2.

Vector Computers

A related model to SIMD is **vector processing**. It is a well-established architecture and compiler model that was popularized by supercomputers and is considerably more widely used than SIMD. Vector processors have high-level operations that work on linear arrays of numbers, or vectors. An example vector operation is

$$A = B \times C$$

vector processor An architecture and compiler model that was popularized by supercomputers in which high-level operations work on linear arrays of numbers.

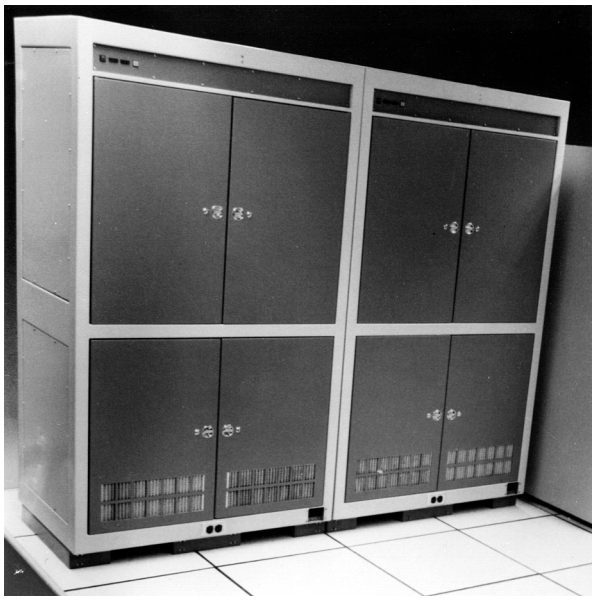


FIGURE 9.11.3 The Goodyear MPP with 16,384 processors. It was delivered May 2, 1983, to NASA Goddard Space Center and was operational the next day. It was decommissioned on March 1, 1991.

where *A, B,* and *C* are each 64-element vectors of 64-bit floating-point numbers. SIMD has similar instructions; the difference is that vector processors depend on pipelined functional units that typically operate on a few vector elements per clock cycle, while SIMD typically operates on all the elements at once.

Advantages of vector computers over traditional SISD processors include the following:

1. Each result is independent of previous results, which enables deep pipelines and high clock rates.
2. A single vector instruction performs a great deal of work, which means fewer instruction fetches in general, and fewer branch instructions and so fewer mispredicted branches.
3. Vector instructions access memory a block at a time, which allows memory latency to be amortized over, say, 64 elements.
4. Vector instructions access memory with known patterns, which allows multiple memory banks to simultaneously supply operands.

These last two advantages mean that vector processors do not need to rely on high hit rates of data caches to have high performance. They tend to rely on low-

latency main memory, often made from SRAM, and have as many as 1024 memory banks to get high memory bandwidth.

Surprisingly, vectors have also found an application in media applications.

Multimedia SIMD

Unfortunately, marketing groups borrowed the SIMD acronym to describe a simple idea: if the data of your application is narrow, rather than performing one operation per clock cycle, use wide registers and ALUs to perform many narrow operations per clock cycle. For example, with 8-bit data and 64-bit registers and ALUs, a processor can perform 8 operations per clock cycle. If the ALU can suppress carries between groups of 8 bits, it can perform narrow adds and subtracts in parallel. Although this use of SIMD is a misnomer, it is very common.

This design is really a subset of a short vector architecture, missing some of the important features and elegance of the full vector design: making the maximum number elements per vector independent of the architecture, strided and index loads and stores, and so on.

Elaboration: Although MISD fills out Flynn's classification, it is difficult to envision. A single instruction stream is simpler than multiple instruction streams, but multiple instruction streams with multiple data streams (MIMD) are easier to imagine than multiple instructions with a single data stream (MISD).

Multiple Instruction Multiple Data Computers

It is difficult to distinguish the first MIMD: arguments for the advantages of parallel execution can be traced back to the 19th century [Menabrea 1842]! Moreover, even the first computer from the Eckert-Mauchly Corporation had duplicate units, in this case to improve reliability.

Two of the best-documented multiprocessor projects were undertaken in the 1970s at Carnegie-Mellon University. The first of these was C.mmp, which consisted of 16 PDP-11s connected by a crossbar switch to 16 memory units. It was among the first multiprocessors with more than a few processors, and it had a shared-memory programming model. Much of the focus of the research in the C.mmp project was on software, especially in the operating systems area. A later machine, Cm*, was a cluster-based multiprocessor with a distributed memory and a nonuniform access time, which made programming even more of a challenge. The absence of caches and long remote access latency made data placement critical.

Although very large mainframes were built with multiple processors in the 1970s, multiprocessors did not become highly successful until the 1980s. Bell [1985] suggests the key to success was that the smaller size of the microprocessor

multicomputer Parallel processors with multiple private addresses.

multiprocessor Parallel processors with a single shared address.

allowed the memory bus to replace the interconnection network hardware, and that portable operating systems meant that parallel processor projects no longer required the invention of a new operating system. He distinguishes parallel processors with multiple private addresses by calling them **multicomputers**, reserving the term **multiprocessor** for machines with a single address space. Thus, Bell would classify a cluster as a multicomputer.

The first bus-connected multiprocessor with snooping caches was the Synapse N+1 in 1984. The mid-1980s saw an explosion in the development of alternative coherence protocols, and Archibald and Baer [1986] provide a good survey and analysis, as well as references to the original papers. The late 1980s saw the introduction of many commercial bus-connected, snooping-cache architectures, including the Silicon Graphics 4D/240, the Encore Multimax, and the Sequent Symmetry.

In the effort to build large-scale multiprocessors, two different directions were explored: message-passing multicomputers and scalable shared-memory multiprocessors. Although there had been many attempts to build mesh- and hypercube-connected multiprocessors, one of the first machines to successfully bring together all the pieces was the Cosmic Cube, built at Caltech [Seitz 1985]. It introduced important advances in routing and interconnect technology and substantially reduced the cost of the interconnect, which helped make the multicomputer viable. Commercial machines with related architectures included the Intel iPSC 860, the Intel Paragon, and the Thinking Machines CM-5. Alas, the market for such machines proved to be much smaller than hoped, and Intel withdrew from the business (with ASCI Red being their last machine) and Thinking Machines no longer exists. Today this space is mostly clusters, such as the IBM RS/6000 SP2 (Figure 9.11.4).

Extending the shared-memory model with scalable cache coherence was done by combining a number of ideas. Directory-based techniques for cache coherence were actually known before snooping cache techniques. In fact, the first cache coherence protocol used directories and was used in the IBM 3081 in 1976. The idea of distributing directories with the memories to obtain a scalable implementation of cache coherence (now called distributed shared memory or DSM) was the basis for the Stanford DASH multiprocessor; it is considered the forerunner of the NUMA computers.

There is a vast amount of information on multiprocessors: conferences, journal papers, and even books appear regularly. One good source is the Supercomputing Conference, held annually since 1988. Two major journals, *Journal of Parallel and Distributed Computing* and the *IEEE Transactions on Parallel and Distributed Systems,* contain largely papers on aspects of parallel computing. Textbooks on parallel computing have been written by Almasi and Gottlieb [1989]; Andrews [1991]; Culler, Singh, and Gupta [1998]; and Hwang [1993]. Pfister's book [1998] is one of the few on clusters.

FIGURE 9.11.4 The IBM RS/6000 SP2 with 256 processors. This distributed-memory machine is built using boards from desktop computers largely unchanged plus a custom switch as the interconnect. In contrast to the SP2, most clusters use an off-the-shelf, switched local area network. Photo courtesy of the Lawrence Livermore National Laboratory.

Elaboration: While it was conceivable to write 100 different programs for 100 different processors in an MIMD machine, in practice this proved to be impossible. Today, MIMD programmers write a single source program and think of the same program running on all processors. This approach is sometimes called *single program multiple data* (SPMD).

Further Reading

Almasi, G. S., and A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA.

A textbook covering parallel computers.

Amdahl, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, Atlantic City, NJ, (April) 483–85.

Written in response to the claims of the Illiac IV, this three-page article describes Amdahl's law and gives the classic reply to arguments for abandoning the current form of computing.

Andrews, G. R. [1991]. *Concurrent Programming: Principles and Practice,* Benjamin/Cummings, Redwood City, CA.

A text that gives the principles of parallel programming.

Archibald, J., and J.-L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* 4:4 (November), 273–98.

Classic survey paper of shared-bus cache coherence protocols.

Arpaci-Dusseau, A., R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson [1997]. "High-performance sorting on networks of workstations," *Proc. ACM SIGMOD/PODS Conference on Management of Data*, Tucson, AZ, May 12–15.

How a world record sort was performed on a cluster, including architecture critique of the workstation and network interface. By April 1, 1997, they pushed the record to 8.6 GB in 1 minute and 2.2 seconds to sort 100 MB.

Bell, C. G. [1985]. "Multis: A new class of multiprocessor computers," *Science* 228 (April 26), 462–67.

Distinguishes shared address and nonshared address multiprocessors based on microprocessors.

Culler, D. E., and J. P. Singh, with A. Gupta [1998]. *Parallel Computer Architecture*, Morgan Kaufmann, San Francisco.

A textbook on parallel computers.

Falk, H. [1976]. "Reaching for the Gigaflop," *IEEE Spectrum* 13:10 (October), 65–70.

Chronicles the sad story of the Illiac IV: four times the cost and less than one-tenth the performance of original goals.

Flynn, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December), 1901–09.

Classic article showing SISD/SIMD/MISD/MIMD classifications.

Hennessy, J., and D. Patterson [2003]. Chapters 6 and 8 in *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers, San Francisco.

A more in-depth coverage of a variety of multiprocessor and cluster topics, including programs and measurements.

Hord, R. M. [1982]. *The Illiac-IV, the First Supercomputer,* Computer Science Press, Rockville, MD.

A historical accounting of the Illiac IV project.

Hwang, K. [1993]. *Advanced Computer Architecture with Parallel Programming*, McGraw-Hill, New York.

Another textbook covering parallel computers.

Kozyrakis, C., and D. Patterson [2003]. "Scalable vector processors for embedded systems," *IEEE Micro* 23:6 (November–December), 36–45.

Examination of a vector architecture for the MIPS instruction set in media and signal processing.

Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," *Bibliothèque Universelle de Genève* (October).

Certainly the earliest reference on multiprocessors, this mathematician made this comment while translating papers on Babbage's mechanical computer.

Pfister, G. F. [1998]. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*, second edition, Prentice-Hall, Upper Saddle River, NJ.

An entertaining book that advocates clusters and is critical of NUMA multiprocessors.

Seitz, C. [1985]. "The Cosmic Cube," *Comm. ACM* 28:1 (January), 22–31.

A tutorial article on a parallel processor connected via a hypertree. The Cosmic Cube is the ancestor of the Intel supercomputers.

Slotnick, D. L. [1982]. "The conception and development of parallel processors—A personal memoir," *Annals of the History of Computing* 4:1 (January), 20–30.

Recollections of the beginnings of parallel processing by the architect of the Illiac IV.

9.12 Exercises

9.1 [15] <§9.1> Write a one-page article examining your life for ways in which concurrency is present and mutual exclusion is obtained. You may want to consider things such as freeways going from two lanes to one, waiting in lines at different types of businesses, obtaining the attention of your instructor to ask questions, and so on. Try to discover different means and mechanisms that are used for both communication and synchronization. Are there any situations in which you wish a different algorithm were used so that either latency or bandwidth were improved, or perhaps the system were more "fair"?

9.2 [10] <§9.1> Consider the following portions of two different programs running at the same time on two processors in a symmetric multiprocessor (SMP). Assume that before this code is run, both `x` and `y` are 0.

Processor 1: `...; x = x + 2; y = x + y; ...`

Processor 2: `...; y = x + 2; ...`

What are the possible resulting values of `x` and `y`, assuming the code is implemented using a load-store architecture? For each possible outcome, explain how `x` and `y` might obtain those values. (Hint: You must examine all of the possible interleavings of the assembly language instructions.)

9.3 [10] <§§9.1–9.3> Imagine that all the employees in a huge company have forgotten who runs the company and can only remember whom they work for. Management is considering whether to issue one of the following two statements:

- "Today every employee should ask his boss who his boss is, then tomorrow ask that person who *his* boss is, and so forth, until you eventually discover who runs the company."

- "Everyone, please write the name of your boss on a sheet of paper. Find out what name the person on your sheet of paper has on *his* sheet of paper, and tomorrow write that name on your sheet of paper before coming to work. Repeat this process until you discover who runs the company."

Write a paragraph describing the difference between these two statements and the resulting outcomes. Explain the relationship between the two alternatives described above and what you have learned about concurrency and interprocess synchronization and communication.

9.4 [5] <§§9.1–9.3> {Ex. 9.3} Analyze the performance of the two algorithms above. Consider companies containing 64 people, 1024 people, or 16,384 people. Can you think of any ways to improve either of the two algorithms or to accomplish the same task even faster?

9.5 [5] <§9.3> Count the number of transactions on the bus for the following sequence of activities involving shared data. Assume that both processors use write-back caches, write-update cache coherency, and a block size of one word. Assume that all the words in both caches are clean.

Step	Processor	Memory activity	Memory address
1	processor 1	write	100
2	processor 2	write	104
3	processor 1	read	100
4	processor 2	read	104
5	processor 1	read	104
6	processor 2	read	100

9.6 [10] <§9.3> False sharing can lead to unnecessary bus traffic and delays. Follow the directions for Exercise 9.5, except change the block size to four words.

9.7 [15] <§9.6> Another possible network topology is a three-dimensional grid. Draw the topology as in Figure 9.6.1 on page 9-28 for 64 nodes. What is the bisection bandwidth of this topology?

9.8 [1 week] <§§9.2–9.6> A parallel processor is typically marketed using programs that can scale performance linearly with the number of processors. Port programs written for one parallel processor to another, and measure their absolute performance and how it changes as you change the number of processors. What changes must be made to improve performance of the ported programs on each machine? What is performance according to each program?

9.9 [1 week] <§§9.2–9.6> Instead of trying to create fair benchmarks, invent programs that make one parallel processor look terrible compared with the others and

also programs that always make one look better than the others. What are the key performance characteristics of each program and machine?

9.10 [1 week] <§§9.2–9.6> Parallel processors usually show performance increases as you increase the number of processors, with the ideal being n times speedup for n processors. The goal of this exercise is to create a biased benchmark that gets worse performance as you add processors. For example, one processor on the parallel processor would run the program fastest, two would be slower, four would be slower than two, and so on. What are the key performance characteristics for each organization that give inverse linear speedup?

9.11 [1 week] <§§9.2–9.6> Networked workstations may be considered parallel processors, albeit with slow communication relative to computation. Port parallel processor benchmarks to a network using remote procedure calls for communication. How well do the benchmarks scale on the network versus the parallel processor? What are the practical differences between networked workstations and a commercial parallel processor?

9.12 [1 week] <§§9.2–9.6> *Superlinear* performance improvement means that a program on n processors is more than n times faster than the equivalent uniprocessor. One argument for superlinear speedup is that time spent servicing interrupts or switching contexts is reduced when you have many processors because only one needs service interrupts and there are more processors to be shared by users. Measure the time spent on a workload in handling interrupts or context switching for a uniprocessor versus a parallel processor. This workload may be a mix of independent jobs for a multiprogramming environment or a single large job. Does the argument hold?

9.13 [15] <§9.10> Construct a scenario whereby a truly revolutionary architecture—pick your favorite candidate—will play a significant role. "Significant" is defined as 10% of the computers sold, 10% of the users, 10% of the money spent on computers, or 10% of some other figure of merit.

9.14 [20] <§§9.1–9.10> This chapter introduced many new vocabulary terms related to the subject of multiprocessors. Some of the exercises in this chapter (e.g., Exercises 9.1 and 9.3) are based on an analogy in which people are thought of as processors and collections of people as multiprocessors. Write a one-page article exploring this analogy in more detail. For example, are there collections of people who use techniques akin to message passing or shared memories? Can you create analogies for cache coherency protocols, network topologies, or clusters? Try to include at least one vocabulary term from each section of the text.