of a tree. Many of the operations performed on highly parallel systems such as BlueGene/P require the participation of all the nodes. For example, consider finding the minimum value of a set of 65,536 values, one held in each node. The collective network joins all the nodes in a tree. Whenever two nodes send their respective values to a higher-level node, it selects out the smallest one and forwards it upward. In this way, far less traffic reaches the root than if all 65,636 nodes sent a message there.

The third network is the barrier network, used to implement global barriers and interrupts. Some algorithms work in phases with each node required to wait until all the others have completed the phase before starting the next phase. The barrier network allows the software to define these phases and provide a way to suspend all compute CPUs that reach the end of a phase until all of them have reached the end, at which time they are all released. Interrupts also use this network.

The fourth and fifth networks both use 10-gigabit Ethernet. One of them connects the I/O nodes to the file servers, which are external to BlueGene/P, and to the Internet beyond. The other one is used for debugging the system.

Each CPU node runs a small, custom, lightweight kernel that supports a single user and a single process. This process has at most four threads, one running on each CPU in the node. This simple structure was designed for high performance and high reliability.

For additional reliability, application software can call a library procedure to make a checkpoint. Once all outstanding messages have been cleared from the network, a global checkpoint can be made and stored so that in the event of a system failure, the job can be restarted from the checkpoint, rather than from the beginning. The I/O nodes run a traditional Linux operating system and support multiple processes.

Work is continuing to develop the next generation of BlueGene system, called the BlueGene/Q. This system is expected to go online in 2012, and it will have 18 processors per compute chip, which also feature simultaneous multithreading. These two features should greatly increase the number of instructions per cycle the system can execute. The system is expected to reach speeds of 20 petaflops/sec. For more information about BlueGene see Adiga et al. (2002), Alam et al., 2008, Almasi et al. (2003a, 2003b), Blumrich et al. (2005), and IBM (2008).

Red Storm

As our second example of an MPP, let us consider the Red Storm machine (also called Thor's Hammer) at Sandia National Laboratory. Sandia is operated by Lockheed Martin and does classified and unclassified work for the U.S. Department of Energy. Some of the classified work concerns the design and simulation of nuclear weapons, which is highly compute intensive.

Sandia has been in this business for a long time and over the decades has had many leading-edge supercomputers over the years. For decades, it favored vector

supercomputers, but eventually technology and economics made MPPs more cost effective. By 2002, the then-current MPP, called ASCI Red, was getting a bit creaky. Although it had 9460 nodes, collectively they had a mere 1.2 TB of RAM and 12.5 TB of disk space, and the system could barely crank out 3 teraflops/sec. So in the summer of 2002, Sandia selected Cray Research, a long-time supercomputer vendor, to build it a replacement for ASCI Red.

The replacement was delivered in August 2004, a remarkably short design and implementation cycle for such a large machine. The reason it could be designed and delivered so quickly is that Red Storm uses almost entirely off-the-shelf parts, except for one custom chip used for routing. In 2006, the system was updated with new processors; we detail this version of Red Storm here.

The CPU selected for Red Storm was the AMD 2.4-GHz dual-core Opteron. The Opteron has several key characteristics that made it the first choice. The first is that it has three operating modes. In legacy mode, it runs standard Pentium binary programs unmodified. In compatibility mode, the operating system runs in 64-bit mode and can address 2^{64} bytes of memory, but application programs run in 32-bit mode. Finally, in 64-bit mode, the entire machine is 64 bits and all programs can address the full 64-bit address space. In 64-bit mode, it is possible to mix and match software: both 32-bit and 64-bit programs can run at the same time, allowing an easy upgrade path.

The Opteron's second key characteristic is its attention to the memory bandwidth problem. In recent years, CPUs have been getting faster and faster and memory has not been keeping pace, resulting in a massive penalty when there is a level 2 cache miss. AMD integrated the memory controller into the Opteron so it can run at the speed of the processor clock instead of the speed of the memory bus, which improves memory performance. The controller can handle eight DIMMS of 4 GB each, for a maximum total memory of 32 GB per Opteron. In the Red Storm system, each Opteron has only 2–4 GB. However, as memory gets cheaper, no doubt more will be added in the future. By utilizing dual-core Opterons, the system was able to double the raw compute power.

Each Opteron has its own dedicated custom network processor called the **Seastar**, manufactured by IBM. The Seastar is a critical component since nearly all the data traffic between the processors goes over the Seastar network. Without the very high-speed interconnect provided by these custom chips, the system would quickly bog down in data.

Although the Opterons are commercially available off the shelf, the Red Storm packaging is custom built. Each Red Storm board contains four Opterons, 4 GB of RAM, four Seastars, a RAS (Reliability, Availability, and Service) processor, and a 100-Mbps Ethernet chip, as shown in Fig. 8-40.

A set of eight boards is plugged into a backplane and inserted into a card cage. Each cabinet holds three card cages for a total of 96 Opterons, plus the necessary power supplies and fans. The full system consists of 108 cabinets for compute nodes, giving a total of 10,368 Opterons (20,736 processors) with 10 TB of

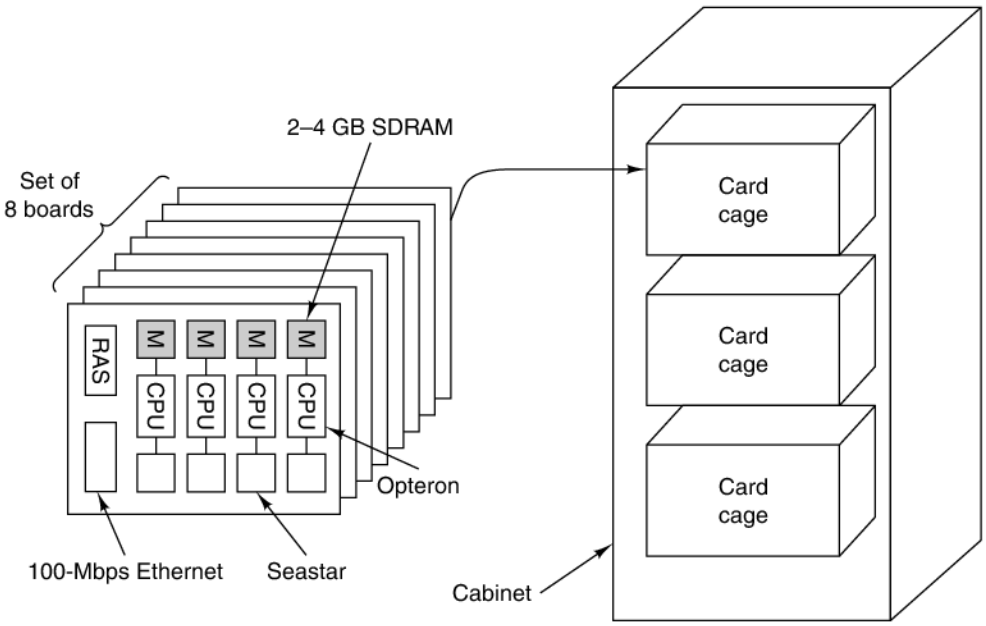

Figure 8-40. Packaging of the Red Storm components.

SDRAM. Each CPU has access only to its own SDRAM. There is no shared memory. The theoretical computing power of the system is 41 teraflops/sec.

The interconnection between the Opteron CPUs is done by the custom Seastar routers, one router per Opteron CPU. They are connected in a 3D torus of size $27 \times 16 \times 24$ with one Seastar at each mesh point. Each Seastar has seven bidirectional 24-Gbps links, going north, east, south, west, up, down, and to the Opteron. The transit time between adjacent mesh points is 2 microsec. Across the entire set of compute nodes it is only 5 microsec. A second network using 100-Mbps Ethernet is used for service and maintenance.

In addition to the 108 compute cabinets, the system also contains 16 cabinets for I/O and service processors. Each cabinet holds 32 Opterons. These 512 CPUs are split: 256 for I/O and 256 for service. The rest of the space is for disks, which are organized as RAID 3 and RAID 5, each with a parity drive and a hot spare. The total disk space is 240 TB. The combined disk bandwidth is 50 GB/sec.

The system is partitioned into classified and unclassified sections, with switches between the parts so they can be mechanically coupled or decoupled. A total of 2688 are always in the classified section and another 2688 Opterons are always in the unclassified section. The remaining 4992 Opterons are switchable into either section, as depicted in Fig. 8-41. The 2688 classified Opterons each have 4 GB of RAM; all the rest have 2 GB each. Apparently classified work is memory intensive. The I/O and service processors are split between the two parts.

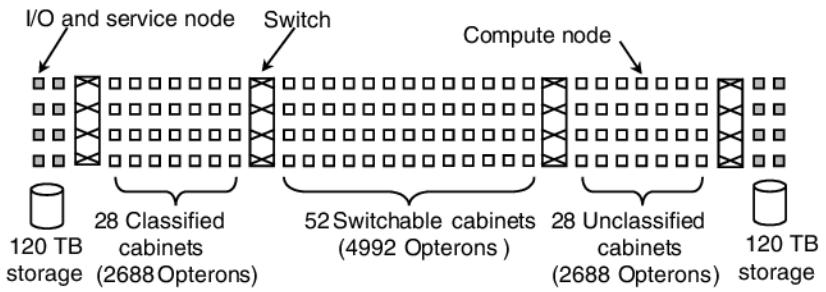

Figure 8-41. The Red Storm system as viewed from above.

Everything is housed in a new 2000-m^2 custom building. The building and site have been designed so that the system can be upgraded to as many as 30,000 Opterons in the future if required. The compute nodes draw 1.6 megawatts of power; the disks draw another megawatt. Adding in the fans and air conditioning, the whole thing uses 3.5 MW.

The computer hardware and software cost $90 million. The building and cooling cost another $9 million, so the total cost came in at just under $100 million, although some of that was nonrecurring engineering cost. If you want to order an exact clone, $60 million would be a good number to keep in mind. Cray intends to sell smaller versions of the system to other government and commercial customers under the name X3T.

The compute nodes run a lightweight kernel called **catamount**. The I/O and service nodes run plain vanilla Linux with a small addition to support MPI (discussed later in this chapter). The RAS nodes run a stripped-down Linux. Extensive software from ASCI Red is available for use on Red Storm, including CPU allocators, schedulers, MPI libraries, math libraries, as well as the application programs.

With such a large system, achieving high reliability is essential. Each board has a RAS processor for doing system maintenance and there are special hardware facilities as well. The goal is an MTBF (Mean Time Between Failures) of 50 hours. ASCI Red had a hardware MTBF of 900 hours but was plagued by an operating-system crash every 40 hours. Although the new hardware is much more reliable than the old, the weak point remains the software.

For more information about Red Storm, see Brightwell et al. (2005, 2010).

A Comparison of BlueGene/P and Red Storm

Red Storm and BlueGene/P are comparable in some ways but different in others, so it is interesting to put some of the key parameters next to each other, as presented in Fig. 8-42.

Item	BlueGene/P	Red Storm
CPU	32-Bit PowerPC	64-Bit Opteron
Clock	850 MHz	2.4 GHz
Compute CPUs	294,912	20,736
CPUs/board	128	8
CPUs/cabinet	4096	192
Compute cabinets	72	108
Teraflops/sec	1000	124
Memory/CPU	512 MB	2–4 GB
Total memory	144 TB	10 TB
Router	PowerPC	Seastar
Number of routers	73,728	10,368
Interconnect	3D torus $72 \times 32 \times 32$	3D torus $27 \times 16 \times 24$
Other networks	Gigabit Ethernet	Fast Ethernet
Partitionable	No	Yes
Compute OS	Custom	Custom
I/O OS	Linux	Linux
Vendor	IBM	Cray Research
Expensive	Yes	Yes

Figure 8-42. A comparison of BlueGene/P and Red Storm.

The two machines were built in the same time frame, so their differences are due not to technology but to designers' different visions and to some extent to the differences between the builders, IBM and Cray. BlueGene/P was designed from the beginning as a commercial machine, which IBM hopes to sell in large numbers to biotech, pharmaceutical, and other companies. Red Storm was built on special contract with Sandia, although Cray plans to make a smaller version for sale, too.

IBM's vision is clear: combine existing cores to produce a custom chip that can be mass produced cheaply, run it at a low speed, and hook together a very large number of them using a modest-speed communication network. Sandia's vision is equally clear, but different: use a powerful off-the-shelf 64-bit CPU, design a very fast custom router chip, and throw in a lot of memory to produce a far more powerful node than BlueGene/P so fewer will be needed and communication between them will be faster.

These decisions had consequences for the packaging. Because IBM built a custom chip combining the processor and router, it achieved a higher packing density: 4096 CPUs/cabinet. Because Sandia went for an unmodified off-the-shelf CPU chip and 2–4 GB of RAM per node, it could get only 192 compute processors in a cabinet. Consequently, Red Storm takes up more floor space and consumes more power than BlueGene/P.

In the exotic world of national laboratory computing, the bottom line is performance. In this respect, BlueGene/P wins, 1000 TF/sec to 124 TF/sec, but Red Storm was designed to be expandable, so by throwing more Opterons at the problem, Sandia could probably up its performance significantly. IBM could respond by cranking the clock up a bit (850 MHz is not really pushing the state-of-the-art very hard). In short, MPP supercomputers have not even come close to any physical limits yet and will continue growing for years to come.

8.4.3 Cluster Computing

The other style of multicomputer is the **cluster computer** (Anderson et al., 1995, and Martin et al., 1997). It typically consists of hundreds or thousands of PCs or workstations connected by a commercially available network board. The difference between an MPP and a cluster is analogous to the difference between a mainframe and a PC. Both have a CPU, both have RAM, both have disks, both have an operating system, and so on. The mainframe just has faster ones (except maybe the operating system). Yet qualitatively they feel different and are used and managed differently. This same difference holds for MPPs vs. clusters.

Historically, the key element that made MPPs special was their high-speed interconnect, but the recent arrival of commercial, off-the-shelf, high-speed interconnects has begun to close the gap. All in all, clusters are likely to drive MPPs into ever tinier niches, just as PCs have turned mainframes into esoteric specialty items. The main niche for MPPs is high-budget supercomputers, where peak performance is everything and if you have to ask the price you cannot afford one.

While many kinds of clusters exist, two species dominate: centralized and decentralized. A centralized cluster is a cluster of workstations or PCs mounted in a big rack in a single room. Sometimes they are packaged in a much more compact way than usual to reduce physical size and cable length. Typically, the machines are homogeneous and have no peripherals other than network cards and possibly disks. Gordon Bell, the designer of the PDP-11 and VAX, has called such machines **headless workstations** (because they have no owners). We were tempted to call them headless COWs, but feared such a term would gore too many holy cows, so we refrained.

Decentralized clusters consist of the workstations or PCs spread around a building or campus. Most of them are idle many hours a day, especially at night. Usually, these are connected by a LAN. Typically, they are heterogeneous and have a full complement of peripherals, although having a cluster with 1024 mice is really not much better than a cluster with 0 mice. Most importantly, many of them have owners who have emotional attachments to their machines and tend to frown upon some astronomer trying to simulate the big bang on theirs. Using idle workstations to form a cluster invariably means having some way to migrate jobs off machines when their owners want to reclaim them. Job migration is possible but adds software complexity.

Clusters are often smallish affairs, ranging from a dozen to perhaps 500 PCs. However, it is also possible to build very large ones from off-the-shelf PCs. Google has done this in an interesting way that we will now look at.

Google

Google is a popular search engine for finding information on the Internet. While its popularity is due, in part, to its simple interface and fast response time, its design is anything but simple. From Google's point of view, the problem is that it has to find, index, and store the entire World Wide Web (an estimated 40 billion pages), be able to search the whole thing in under 0.5 sec, and handle tens of thousands of queries/sec coming from all over the world 24 hours a day. In addition, it must never go down, not even in the face of earthquakes, electrical power failures, telecom outages, hardware failures and software bugs. And, of course, it has to do all of this as cheaply as possible. Building a Google clone is definitely not an exercise for the reader.

How does Google do it? To start with, Google operates multiple data centers around the world. Not only does this approach provide backups in case one of them is swallowed by an earthquake, but when *www.google.com* is looked up, the sender's IP address is inspected and the address of the nearest data center is supplied. The browser then sends the query there.

Each data center has at least one OC-48 (2.488-Gbps) fiber-optics connection to the Internet, on which it receives queries and sends answers, as well as an OC-12 (622-Mbps) backup connection from a different telecom provider, in case the primary ones go down. Uninterruptable power supplies and emergency diesel generators are available at all data centers to keep the show going during power failures. Consequently, during a major natural disaster, performance will suffer, but Google will keep running.

To get a better understanding of why Google chose the architecture it did, it is useful to briefly describe how a query is processed once it hits its designated data center. After arriving at the data center (step 1 in Fig. 8-43), the load balancer routes the query to one of the many query handlers (2), and to the spelling checker (3) and ad server (4) in parallel. Then the search words are looked up on the index servers (5) in parallel. These servers contain an entry for each word on the Web. Each entry lists all the documents (Web pages, PDF files, PowerPoint presentations, etc.) containing the word, sorted in page-rank order. Page rank is determined by a complicated (and secret) formula, but the number of links to a page and their own ranks play a large role.

To get higher performance, the index is divided into pieces called **shards** that can be searched in parallel. Conceptually, at least, shard 1 contains all the words in the index, with each one followed by the IDs of the n highest-ranked documents containing that word. Shard 2 contains all the words and the IDs of the n next-

highest-ranked documents, and so on. As the Web grows, each of these shards may later be split with the first k words in one set of shards, the next k words in a second set of shards and so forth, in order to achieve even more search parallelism.

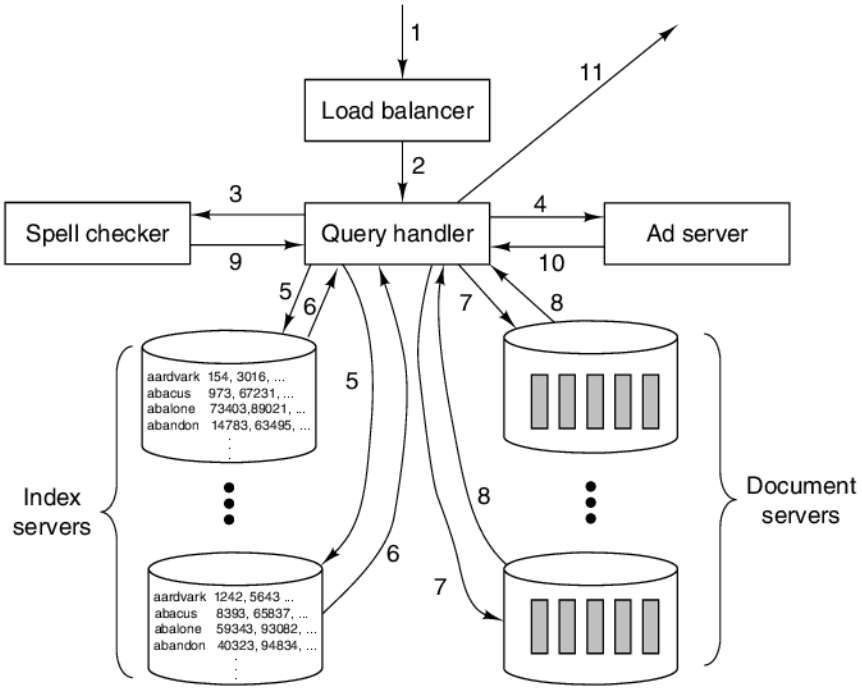

Figure 8-43. Processing of a Google query.

The index servers return a set of document identifiers (6) that are then combined according to the Boolean properties of the query. For example, if the search was for +digital +capybara +dance, then only document identifiers appearing in all three sets are used in the next step. In this step (7), the documents themselves are referenced to extract their titles, URLs, and snippets of text surrounding the search terms. The document servers contain many copies of the entire Web at each data center, hundreds of terabytes at present. The documents are also divided into shards to enhance parallel search. While processing a query does not require reading the whole Web (or even reading the tens of terabytes on the index servers), having to process 100 MB per query is normal.

When the results are returned to the query handler (8), the pages found are collated into page-rank order. If potential spelling errors are detected (9), they are announced and relevant ads are added (10). Displaying ads for advertisers interested in buying specific search terms (e.g., "hotel" or "camcorder") is how Google makes its money. Finally, the results are formatted in HTML (HyperText Markup Language) and sent to the user as a Web page.

With this background, we can now examine the Google architecture. Most companies, when faced with a huge data base, massive transaction rate, and the need for high reliability, would buy the biggest, fastest, and most reliable equipment on the market. Google did just the opposite. It bought cheap, modest-performance PCs. Lots of them. And with them, it built the world's largest off-the-shelf cluster. The driving principle behind this decision was simple: optimize price/performance.

The logic behind this decision lies in economics: commodity PCs are very cheap. High-end servers are not and large multiprocessors are even less so. Thus while a high-end server might have two or three times the performance of a midrange desktop PC, it will typically be 5–10 times the price, which is not cost effective.

Of course, cheap PCs fail more often than top-of-the-line servers, but the latter fail, too, so the Google software had to be designed to work with failing hardware no matter what kind of equipment it was using. Once the fault-tolerance software was written, it did not really matter whether the failure rate was 0.5% per year or 2% per year, failures had to be dealt with. Google's experience is that about 2% of the PCs fail each year. More than half the failures are due to faulty disks, followed by power supplies and then RAM chips. After burn-in, CPUs never fail. Actually, the biggest source of crashes is not hardware at all; it is software. The first response to a crash is just to reboot, which often solves the problem (the electronic equivalent of the doctor saying: "Take two aspirins and go to bed.").

A typical modern Google PC consists of a 2-GHz Intel processor, 4 GB of RAM, and a disk of around 2 TB, the kind of thing a grandmother might buy for checking her email occasionally. The only specialty item is an Ethernet chip. Not exactly state of the art, but very cheap. The PCs are housed in 1u-high cases (about 5 cm thick) and stacked 40 high in 19-inch racks, one stack in front and one stack in back for a total of 80 PCs per rack. The PCs in a rack are connected by switched Ethernet, with the switch inside the rack. The racks in a data center are also connected by switched Ethernet, with two redundant switches per data center used to survive switch failures.

The layout of a typical Google data center is illustrated in Fig. 8-44. The incoming high-bandwidth OC-48 fiber is routed to each of two 128-port Ethernet switches. Similarly, the backup OC-12 fiber is also routed to each of the two switches. The incoming fibers use special input cards and do not occupy any of the 128 Ethernet ports.

Each rack has four Ethernet links coming out of it, two to the left switch and two to the right switch. In this configuration, the system can survive the failure of either switch. Since each rack has four connections to the switch (two from the front 40 PCs and two from the back 40 PCs), it takes four link failures or two link failures and a switch failure to take a rack offline. With a pair of 128-port switches and four links from each rack, up to 64 racks can be supported. With 80 PCs per rack, a data center can have up to 5120 PCs. But, of course, racks do not have to

hold exactly 80 PCs and switches can be larger or smaller than 128 ports; these are just typical values for a Google cluster.

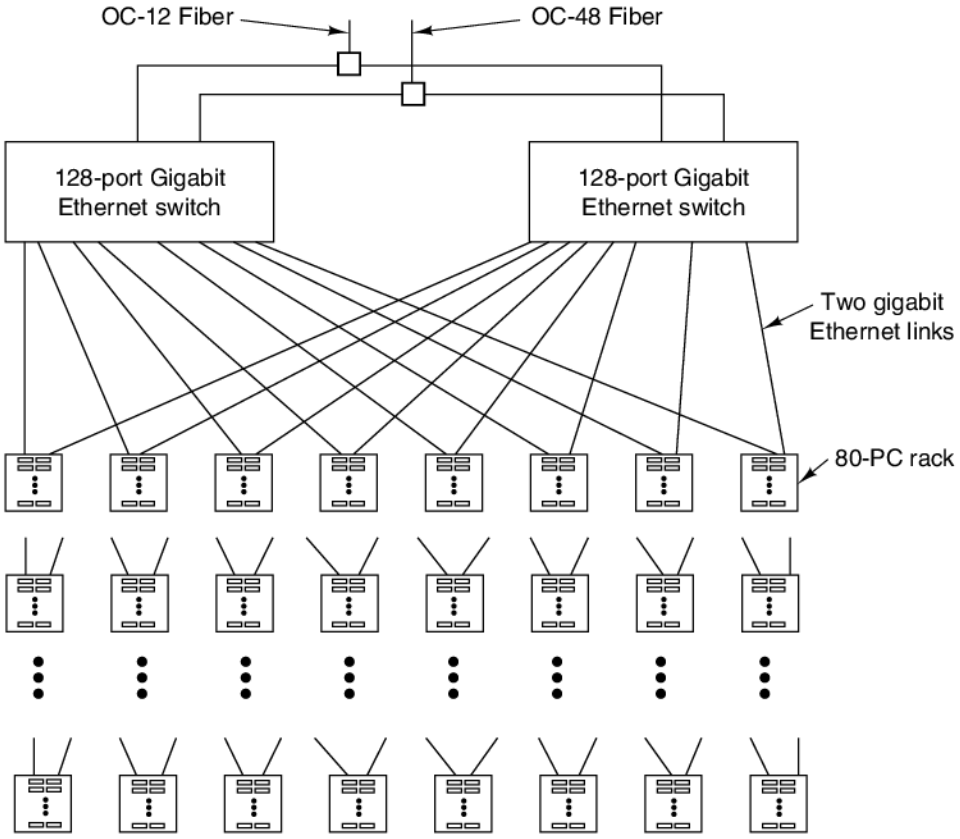

Figure 8-44. A typical Google cluster.

Power density is also a key issue. A typical PC burns about 120 watts or about 10 kW per rack. A rack needs about 3 m^2 so that maintenance personnel can install and remove PCs and for the air conditioning to function. These parameters give a power density of over 3000 watts/m^2. Most data centers are designed for 600–1200 watts/m^2, so special measures are required to cool the racks.

Google has learned three key things about running massive Web servers that bear repeating.

1. Components will fail so plan for it.
2. Replicate everything for throughput and availability.
3. Optimize price/performance.

The first item says that you need to have fault-tolerant software. Even with the best of equipment, when you have a massive number of components, some will fail and the software has to be able to handle it. Whether you have one failure a week or two, the software has to be able to handle failures.

The second item points out that both hardware and software have to be highly redundant. Doing so not only improves the fault-tolerance properties, but also the throughput. In the case of Google, the PCs, disks, cables, and switches are all replicated many times over. Furthermore, the index and the documents are broken into shards and the shards are heavily replicated in each data center and the data centers are themselves replicated.

The third item is a consequence of the first two. If the system has been properly designed to deal with failures, buying expensive components such as RAIDs with SCSI disks is a mistake. Even they will fail, but spending 10 times as much to cut the failure rate in half is a bad idea. Better to buy 10 times as much hardware and deal with the failures when they occur. At the very least, having more hardware will give better performance when everything is working.

For more information about Google, see Barroso et al. (2003), and Ghemawat et al. (2003).

8.4.4 Communication Software for Multicomputers

Programming a multicomputer requires special software, usually libraries, for handling interprocess communication and synchronization. In this section we will say a few words about this software. For the most part, the same software packages run on MPPs and clusters, so applications can be easily ported between platforms.

Message-passing systems have two or more processes running independently of one another. For example, one process may be producing some data and one or more others may be consuming it. There is no guarantee that when the sender has more data the receiver(s) will be ready for it, as each one runs its own program.

Most message-passing systems provide two primitives (usually library calls), send and receive, but several different kinds of semantics are possible. The three main variants are

1. Synchronous message passing.
2. Buffered message passing.
3. Nonblocking message passing.

In **synchronous message passing**, if the sender executes a send and the receiver has not yet executed a receive, the sender is blocked (suspended) until the receiver executes a receive, at which time the message is copied. When the sender gets control back after the call, it knows that the message has been sent and correctly received. This method has the simplest semantics and does not require any

buffering. It has the severe disadvantage that the sender remains blocked until the receiver has gotten and acknowledged receipt of the message.

In **buffered message passing**, when a message is sent before the receiver is ready, the message is buffered somewhere, for example, in a mailbox, until the receiver takes it out. Thus in buffered message passing, a sender can continue after a send, even if the receiver is busy with something else. Since the message has actually been sent, the sender is free to reuse the message buffer immediately. This scheme reduces the time the sender has to wait. Basically, as soon as the system has sent the message the sender can continue. However, the sender now has no guarantee that the message was correctly received. Even if communication is reliable, the receiver may have crashed before getting the message.

In **nonblocking message passing**, the sender is allowed to continue immediately after making the call. All the library does is tell the operating system to do the call later, when it has time. As a consequence, the sender is hardly blocked at all. The disadvantage of this method is that when the sender continues after the send, it may not reuse the message buffer as the message may not yet have been sent. Somehow it has to find out when it can reuse the buffer. One idea is to have it poll the system to ask. The other is to get an interrupt when the buffer is available. Neither of these makes the software any simpler.

Below we will briefly discuss a popular message-passing system available on many multicomputers: MPI.

MPI—Message-Passing Interface

For quite a few years, the most popular communication package for multicomputers was **PVM** (**Parallel Virtual Machine**) (Geist et al., 1994, and Sunderram, 1990). In recent years it has been largely replaced by **MPI** (**Message-Passing Interface**). MPI is much richer and more complex than PVM, with many more library calls, many more options, and many more parameters per call. The original version of MPI, now called MPI-1, was augmented by MPI-2 in 1997. Below we will give a very cursory introduction to MPI-1 (which contains all the basics), then say a little about what was added in MPI-2. For more information about MPI, see Gropp et al. (1994) and Snir et al. (1996).

MPI-1 does not deal with process creation or management, as PVM does. It is up to the user to create processes using local system calls. Once they have been created, they are arranged into static, unchanging process groups. It is with these groups that MPI works.

MPI is based on four major concepts: communicators, message data types, communication operations, and virtual topologies. A **communicator** is a process group plus a context. A context is a label that identifies something, such as a phase of execution. When messages are sent and received, the context can be used to keep unrelated messages from interfering with one another.

Messages are typed and many data types are supported, including characters, short, regular, and long integers, single- and double-precision floating-point numbers, and so on. It is also possible to construct other types derived from these.

MPI supports an extensive set of communication operations. The most basic one is used to send messages as follows:

```
MPI_Send(buffer, count, data_type, destination, tag, communicator)
```

This call sends a buffer with *count* number of items of the specified data type to the destination. The *tag* field labels the message so the receiver can say it only wants to receive a message with that tag. The last field tells which process group the destination is in (the *destination* field is just an index into the list of processes for the specified group). The corresponding call for receiving a message is

```
MPI_Recv(&buffer, count, data_type, source, tag, communicator, &status)
```

which announces that the receiver is looking for a message of a certain type from a certain source with a certain tag.

MPI supports four basic communication modes. Mode 1 is synchronous, in which the sender may not begin sending until the receiver has called MPI_Recv. Mode 2 is buffered, in which this restriction does not hold. Mode 3 is standard, which is implementation dependent and can be either synchronous or buffered. Mode 4 is ready, in which the sender claims the receiver is available (as in synchronous), but no check is made. Each of these primitives comes in a blocking and a nonblocking version, leading to eight primitives in all. Receiving has only two variants: blocking and nonblocking.

MPI supports collective communication, including broadcast, scatter/gather, total exchange, aggregation, and barrier. For all forms of collective communication, all the processes in a group must make the call and with compatible arguments. Failure to do this is an error. A typical form of collective communication is for processes organized in a tree, in which values propagate up from the leaves to the root, undergoing some processing at each step, for example, adding up the values or taking the maximum.

A basic concept in MPI is the **virtual topology**, in which the processes can be arranged in a tree, ring, grid, torus, or other topology by the user per application. Such an arrangement provides a way to name communication paths and facilitates communication.

MPI-2 adds dynamic processes, remote memory access, nonblocking collective communication, scalable I/O support, real-time processing, and many other new features that are beyond the scope of this book. In the scientific community, a battle raged for years between the MPI and PVM camps. The PVM side said that PVM was easier to learn and simpler to use. The MPI side said the MPI does more and also points out that MPI is a formal standard with a standardization committee and an official defining document. The PVM side agreed but claimed that the lack

of a full-blown standardization bureaucracy is not necessarily a drawback. When all was said and done, it appears that MPI won.

8.4.5 Scheduling

MPI programmers can easily create jobs requesting multiple CPUs and running for substantial periods of time. When multiple independent requests are available from different users, each needing a different number of CPUs for different time periods, the cluster needs a scheduler to determine which job gets to run when.

In the simplest model, the job scheduler requires each job to specify how many CPUs it needs. Jobs are then run in strict FIFO order, as shown in Fig. 8-45(a). In this model, after a job is started, a check is made to see if enough CPUs are available to start the next job in the input queue. If so, it is started, and so on. Otherwise, the system waits until more CPUs become available. As an aside, although we have suggested that this cluster has eight CPUs, it might well have 128 CPUs that are allocated in units of 16 (giving eight CPU groups), or another combination.

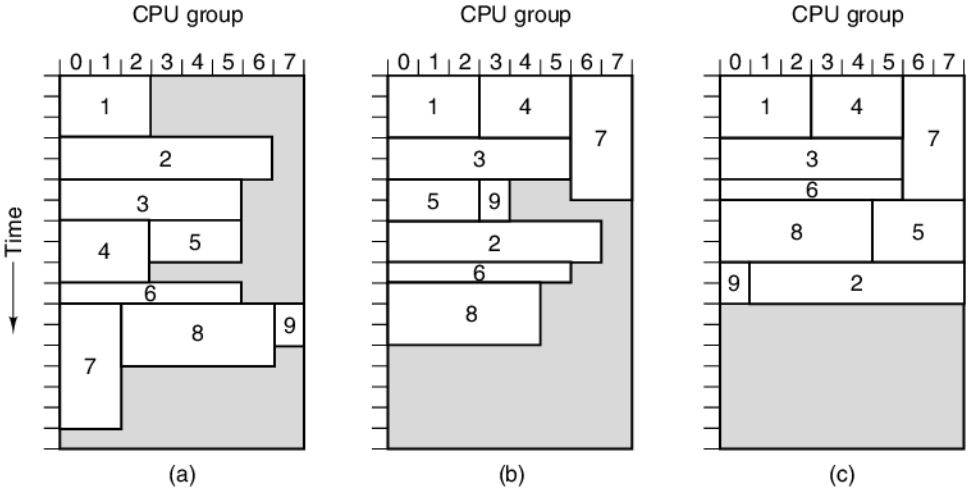

Figure 8-45. Scheduling a cluster. (a) FIFO. (b) Without head-of-line blocking. (c) Tiling. The shaded areas indicate idle CPUs.

A better scheduling algorithm avoids head-of-line blocking by skipping over jobs that do not fit and picking the first one that does fit. Whenever a job finishes, the queue of remaining jobs is checked in FIFO order. This algorithm gives the result of Fig. 8-45(b).

A still more sophisticated scheduling algorithm requires each submitted job to specify its shape, that is, how many CPUs for how many minutes. With that information, the job scheduler can attempt to tile the CPU-time rectangle. Tiling is

especially effective when jobs are submitted during the daytime for execution at night, so the job scheduler has all the information about all the jobs in advance and can run them in optimal order, as illustrated in Fig. 8-45(c).

8.4.6 Application-Level Shared Memory

That multicomputers scale to larger sizes than multiprocessors should be clear from our examples. This reality has led to the development of message-passing systems like MPI. Many programmers do not like this model and would like to have the illusion of shared memory, even if it is not really there. Achieving this goal would be the best of both worlds: large, inexpensive hardware (at least, per node) plus ease of programming. This is the holy grail of parallel computing.

Many researchers have concluded that while shared memory at the architectural level may not scale well, there may be other ways to achieve the same goal. From Fig. 8-21, we see that there are other levels at which a shared memory can be introduced. In the following sections, we will look at some ways that shared memory can be introduced into the programming model on a multicomputer, without it being present at the hardware level.

Distributed Shared Memory

One class of application-level shared-memory system is the page-based system. It goes under the name of **DSM** (**Distributed Shared Memory**). The idea is simple: a collection of CPUs on a multicomputer share a common paged virtual address space. In the simplest version, each page is held in the RAM of exactly one CPU. In Fig. 8-46(a), we see a shared virtual address space consisting of 16 pages, spread over four CPUs.

When a CPU references a page in its own local RAM, the read or write just happens without any further delay. However, when a CPU references a page in a remote memory, it gets a page fault. Instead of having the missing page being brought in from disk, though, the run-time system or operating system sends a message to the node holding the page to unmap it and send it over. After it has arrived, it is mapped in and the faulting instruction restarted, just as with a normal page fault. In Fig. 8-46(b), we see the situation after CPU 0 has faulted on page 10: it is moved from CPU 1 to CPU 0.

This basic idea was first implemented in IVY (Li and Hudak, 1989). It provides a fully shared, sequentially consistent memory on a multicomputer. However, many optimizations are possible to improve the performance. The first optimization, present in IVY, is to allow pages that are marked as read-only to be present at multiple nodes at the same time. Thus when a page fault occurs, a copy of the page is sent to the faulting machine, but the original stays where it is since there is no danger of conflicts. The situation of two CPUs sharing a read-only page (page 10) is illustrated in Fig. 8-46(c).

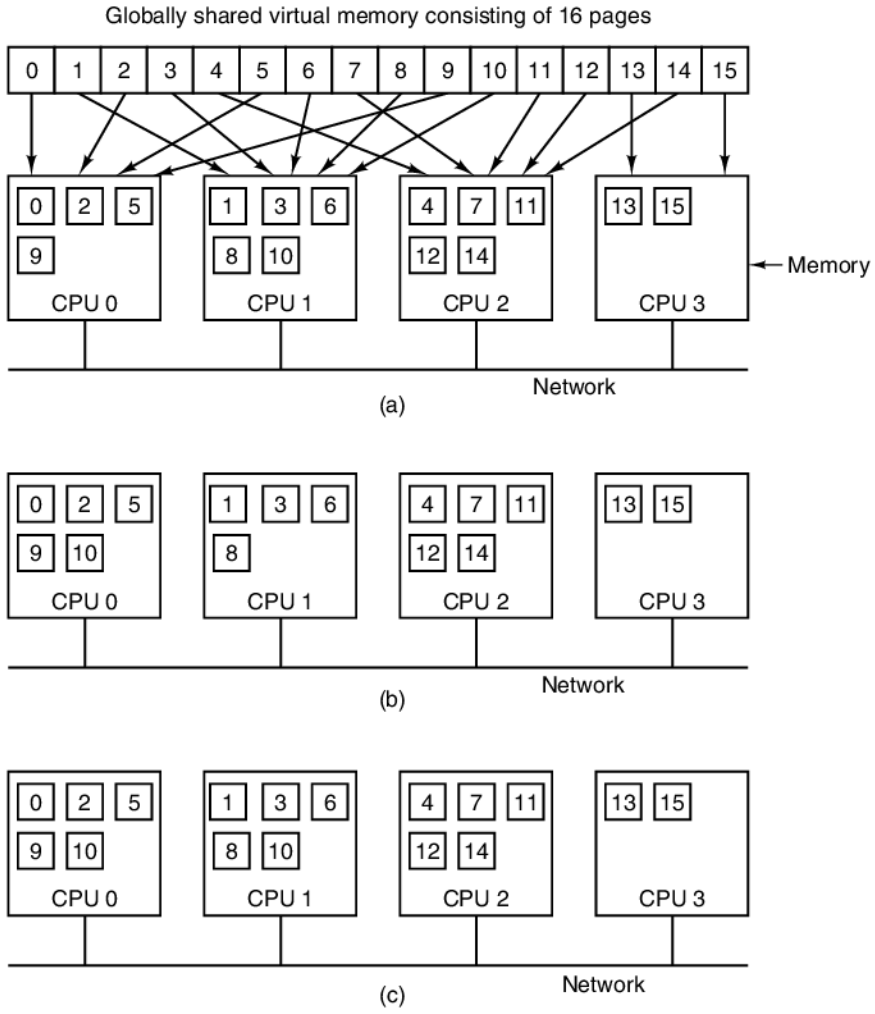

Figure 8-46. A virtual address space consisting of 16 pages spread over four nodes of a multicomputer. (a) The initial situation. (b) After CPU 0 references page 10. (c) After CPU 1 references page 10, here assumed to be a read-only page.

Even with this optimization, performance is frequently unacceptable, especially when one process is actively writing a few words at the top of some page and another process on a different CPU is actively writing a few words at the bottom of the page. Since only one copy of the page exists, the page must be ping-ponged back and forth constantly, a situation known as **false sharing**.

The problem of false sharing can be attacked in several ways. In the Treadmarks system, for example, sequentially consistent memory is abandoned in favor

of release consistency (Amza, 1996). Potentially writable pages may be present at multiple nodes at the same time, but before doing a write, a process must first do an acquire operation to signal its intention. At that point, all copies but the most recent one are invalidated. No other copies may be made until the corresponding release is done, at which time the page can be shared again.

A second optimization done in Treadmarks is to initially map each writable page in read-only mode. When the page is first written to, a protection fault occurs and the system makes a copy of the page, called the **twin**. Then the original page is mapped in as read-write and subsequent writes can go at full speed. When a remote page fault happens later and the page has to be shipped over there, a word-by-word comparison is done between the current page and the twin. Only those words that have been changed are sent, reducing the size of the messages.

When a page fault occurs, the missing page has to be located. Various solutions are possible, including those used in NUMA and COMA machines, such as (home-based) directories. In fact, many of the solutions used in DSM are also applicable to NUMA and COMA because DSM is really just a software implementation of NUMA or COMA with each page being treated like a cache line.

DSM is a hot area of research. Interesting systems include CASHMERE (Kontothanassis, et al., 1997 and Stets et al., 1997), CRL (Johnson et al., 1995), Shasta (Scales et al., 1996), and Treadmarks (Amza, 1996 and Lu et al., 1997).

Linda

Page-based DSM systems like IVY and Treadmarks use the MMU hardware to trap accesses to missing pages. While making and sending differences instead of whole pages helps, the fact remains that pages are an unnatural unit for sharing, so other approaches have been tried.

One such approach is Linda, which provides processes on multiple machines with a highly structured distributed shared memory (Carriero and Gelernter, 1989). This memory is accessed through a small set of primitive operations that can be added to existing languages, such as C and FORTRAN, to form parallel languages, in this case, C-Linda and FORTRAN-Linda.

The unifying concept behind Linda is that of an abstract **tuple space**, which is global to the entire system and accessible to all processes in it. Tuple space is like a global shared memory, only with a certain built-in structure. The tuple space contains some number of **tuples**, each consisting of one or more fields. For C-Linda, field types include integers, long integers, and floating-point numbers, as well as composite types such as arrays (including strings) and structures (but not other tuples). Figure 8-47 shows three tuples as examples.

Four operations are provided on tuples. The first one, out, puts a tuple into the tuple space. For example,

```
out("abc", 2, 5);
```

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is sister", Carolyn, Elinor)
```

Figure 8-47. Three Linda tuples.

puts the tuple ("abc", 2, 5) into the tuple space. The fields of out are normally constants, variables, or expressions, as in

```
out("matrix–1", i, j, 3.14);
```

which outputs a tuple with four fields, the second and third of which are determined by the current values of the variables *i* and *j*.

Tuples are retrieved from the tuple space by the in primitive. They are addressed by content rather than by name or address. The fields of in can be expressions or formal parameters. Consider, for example,

```
in("abc", 2, ? i);
```

This operation "searches" the tuple space for a tuple consisting of the string "abc", the integer, 2, and a third field containing any integer (assuming that *i* is an integer). If found, the tuple is removed from the tuple space and the variable *i* is assigned the value of the third field. The matching and removal are atomic, so if two processes execute the same in operation simultaneously, only one of them will succeed, unless two or more matching tuples are present. The tuple space may even contain multiple copies of the same tuple.

The matching algorithm used by in is straightforward. The fields of the in primitive, called the **template**, are (conceptually) compared to the corresponding fields of every tuple in the tuple space. A match occurs if the following three conditions are all met:

1. The template and the tuple have the same number of fields.
2. The types of the corresponding fields are equal.
3. Each constant or variable in the template matches its tuple field.

Formal parameters, indicated by a question mark followed by a variable name or type, do not participate in the matching (except for type checking), although those containing a variable name are assigned after a successful match.

If no matching tuple is present, the calling process is suspended until another process inserts the needed tuple, at which time the called is automatically revived and given the new tuple. The fact that processes block and unblock automatically means that if one process is about to output a tuple and another is about to input it, it does not matter which goes first.

In addition to out and in, Linda also has a primitive read, which is the same as in except that it does not remove the tuple from the tuple space. There is also a primitive eval, which causes its parameters to be evaluated in parallel and the resulting tuple to be deposited in the tuple space. This mechanism can be used to perform an arbitrary computation. This is how parallel processes are created in Linda.

A common programming paradigm in Linda is the **replicated worker model**. This model is based on the idea of a **task bag** full of jobs to be done. The main process starts out by executing a loop containing

```
out("task-bag", job);
```

in which a different job description is output to the tuple space on each iteration. Each worker starts out by getting a job-description tuple using

```
in("task-bag", ?job);
```

which it then carries out. When it is done, it gets another. New work may also be put into the task bag during execution. In this simple way, work is dynamically divided among the workers, and each worker is kept busy all the time, all with relatively little overhead.

Various implementations of Linda on multicomputer systems exist. In all of them, a key issue is how to distribute the tuples among the machines and how to locate them when needed. Various possibilities include broadcasting and directories. Replication is also an important issue. These points are discussed in Bjornson (1993).

Orca

A somewhat different approach to application-level shared memory on a multicomputer is to use objects instead of just tuples as the unit of sharing. An object consists of internal (hidden) state plus methods for operating on that state. By not allowing the programmer to access the state directly, many possibilities are opened to allow sharing over machines that do not have physical shared memory.

One object-based system that gives the illusion of shared memory on multicomputer systems is Orca (Bal, 1991, Bal et al., 1992, and Bal and Tanenbaum, 1988). Orca is a traditional programming language (based on Modula 2) to which two new features have been added: objects and the ability to create new processes. An Orca object is an abstract data type, analogous to an object in Java or a package in Ada. It encapsulates internal data structures and user-written methods, called **operations**. Objects are passive, that is, they do not contain threads to which messages can be sent. Instead, processes access an object's internal data by invoking its methods.

Each Orca method consists of a list of (guard, block-of-statements) pairs. A guard is a Boolean expression that does not contain any side effects, or the empty

guard, which is the same as the value *true*. When an operation is invoked, all of its guards are evaluated in an unspecified order. If all of them are *false*, the invoking process is delayed until one becomes *true*. When a guard is found that evaluates to *true*, the block of statements following it is executed. Figure 8-48 depicts a *stack* object with two operations, *push* and *pop*.

```
Object implementation stack;
   top:integer;                          # storage for the stack
   stack: array [integer 0..N–1] of integer;

   operation push(item: integer);        # function returning nothing
   begin
      guard top < N – 1 do
         stack[top] := item;             # push item onto the stack
         top := top + 1;                 # increment the stack pointer
      od;
   end;

   operation pop( ): integer;            # function returning an integer
   begin
      guard top > 0 do                   # suspend if the stack is empty
         top := top – 1;                 # decrement the stack pointer
         return stack[top];              # return the top item
      od;
   end;

begin
   top := 0;                             # initialization
end;
```

Figure 8-48. A simplified ORCA stack object, with internal data and two operations.

Once a *stack* has been defined, variables of this type can be declared, as in

```
s, t: stack;
```

which creates two stack objects and initializes the *top* variable in each to 0. The integer variable *k* can be pushed onto the stack *s* by the statement

```
s$push(k);
```

and so forth. The *pop* operation has a guard, so an attempt to pop a variable from an empty stack will suspend the called until another process has pushed something on the stack.

Orca has a **fork** statement to create a new process on a user-specified processor. The new process runs the procedure named in the **fork** statement. Parameters, including objects, may be passed to the new process, which is how objects become distributed among machines. For example, the statement

```
for i in 1 .. n do fork foobar(s) on i; od;
```

generates one new process on each of machines 1 through n, running the program *foobar* in each of them. As these n new processes (and the parent) execute in parallel, they can all push and pop items onto the shared stack s as though they were all running on a shared-memory multiprocessor. It is the job of the run-time system to sustain the illusion of shared memory where it really does not exist.

Operations on shared objects are atomic and sequentially consistent. The system guarantees that if multiple processes perform operations on the same shared object nearly simultaneously, the system chooses some order and all processes see the same order of events.

Orca integrates shared data and synchronization in a way not present in page-based DSM systems. Two kinds of synchronization are needed in parallel programs. The first kind is mutual-exclusion synchronization, to keep two processes from executing the same critical region at the same time. In Orca, each operation on a shared object is effectively like a critical region because the system guarantees that the final result is the same as if all the critical regions were executed one at a time (i.e., sequentially). In this respect, an Orca object is like a distributed form of a monitor (Hoare, 1975).

The other kind of synchronization is condition synchronization, in which a process blocks waiting for some condition to hold. In Orca, condition synchronization is done with guards. In the example of Fig. 8-48, a process trying to pop an item from an empty stack will be suspended until the stack is no longer empty. After all, you cannot pop a word from an empty stack.

The Orca run-time system handles object replication, migration, consistency, and operation invocation. Each object can be in one of two states: single copy or replicated. An object in single-copy state exists on only one machine, so all requests for it are sent there. A replicated object is present on all machines containing a process using it, which makes read operations easier (since they can be done locally), at the expense of making updates more expensive. When an operation that modifies a replicated object is executed, it must first get a sequence number from a centralized process that issues them. Then a message is sent to each machine holding a copy of the object, telling it to execute the operation. Since all such updates bear sequence numbers, all machines just carry out the operations in sequence order, which guarantees sequential consistency.

8.4.7 Performance

The point of building a parallel computer is to make it go faster than a uniprocessor machine. If it does not achieve that simple goal, it is not worth having. Furthermore, it should achieve the goal in a cost-effective manner. A machine that is twice as fast as a uniprocessor at 50 times the cost is not likely to be a big seller. In this section we will examine some of the performance issues associated with parallel computer architectures, starting with how you even measure it.

Hardware Metrics

From a hardware perspective, the performance metrics of interest are the CPU and I/O speeds and the performance of the interconnection network. The CPU and I/O speeds are the same as in the uniprocessor case, so the key parameters of interest in a parallel system are those associated with the interconnect. There are two key items: latency and bandwidth, which we will now look at in turn.

The roundtrip latency is the time it takes for a CPU to send a packet and get a reply. If the packet is sent to a memory, then the latency measures the time to read or write a word or block of words. If it is sent to another CPU, it measures the interprocessor communication time for packets of that size. Usually, the latency of interest is for minimal packets, often one word or a small cache line.

The latency is built up from several factors and is different for circuit-switched, store-and-forward, virtual cut through, and wormhole-routed interconnects. For circuit switching, the latency is the sum of the setup time and the transmission time. To set up a circuit, a probe packet has to be sent out to reserve the resources and then report back. Once that has happened, the data packet has to be assembled. When it is ready, bits can flow at full speed, so if the total setup time is T_s, the packet size is p bits, and the bandwidth b bits/sec, the one-way latency is $T_s + p/b$. If the circuit is full duplex, then there is no setup time for the reply, so the minimum latency for sending a p-bit packet and getting a p-bit reply is $T_s + 2p/b$ sec.

For packet switching, it is not necessary to send a probe packet to the destination in advance, but there is still some internal setup time to assemble the packet, T_a. Here the one-way transmission time is $T_a + p/b$, but this is only the time to get the packet into the first switch. There is a finite delay within the switch, say T_d and then the process is repeated to the next switch and so on. The T_d delay is composed of both processing time and queueing delay, waiting for the output port to become free. If there are n switches, then the total one-way latency is given by the formula $T_a + n(p/b + T_d) + p/b$, where the final term is due to the copy from the last switch to the destination.

The one-way latencies for virtual cut through and wormhole routing in the best case are close to $T_a + p/b$ because there is no probe packet to set up a circuit, and no store-and-forward delay either. Basically, it is the initial setup time to assemble the packet, plus the time to push the bits out the door. In all cases, propagation delay has to be added, but that is usually small.

The other hardware metric is bandwidth. Many parallel programs, especially in the natural sciences, move a lot of data around, so the number of bytes/sec that the system can move is critical to performance. Several metrics for bandwidth exist. We have seen one of them—bisection bandwidth—already. Another is **aggregate bandwidth**, which is computed by simply adding up the capacities of all the links. This number gives the maximum number of bits that can be in transit at once. Yet another important metric is the average bandwidth out of each CPU.

If each CPU is capable of outputting 1 MB/sec, it does little good that the interconnect has a bisection bandwidth of 100 GB/sec. Communication will be limited by how much data each CPU can output.

In practice, actually achieving anything even close to the theoretical bandwidth is very difficult. Many sources of overhead work to reduce the capacity. For example, there is always some per-packet overhead associated with each packet: assembling it, building its header, and getting it going. Sending 1024 4-byte packets will never achieve the same bandwidth as sending one 4096-byte packet. Unfortunately, for achieving low latencies, using small packets is better, since large ones block the lines and switches too long. Thus there is an inherent conflict between achieving low average latencies and high-bandwidth utilization. For some applications, one is more important than the other and for other applications it is the other way around. It is worth noting, however, that you can always buy more bandwidth (by putting in more or wider wires), but you cannot buy lower latencies. Thus it is generally better to err on the side of making latencies as short as possible, and worry about bandwidth later.

Software Metrics

Hardware metrics like latency and bandwidth look at what the hardware is capable of doing. However, users have a different perspective. They want to know how much faster their programs are going to run on a parallel computer than on a uniprocessor. For them, the key metric is speed-up: how much faster a program runs on an n-processor system than on a one-processor system. Typically these results are shown in graphs like those of Fig. 8-49. Here we see several different parallel programs run on a multicomputer consisting of 64 Pentium Pro CPUs. Each curve shows the speed-up of one program with k CPUs as a function of k. Perfect speed-up is indicated by the dotted line, in which using k CPUs makes the program go k times faster, for any k. Few programs achieve perfect speed-up, but some come close. The N-body problem parallelizes extremely well; awari (an African board game) does reasonably well; but inverting a certain skyline matrix does not go more than five times faster no matter how many CPUs are available. The programs and results are discussed in Bal et al. (1998).

Part of the reason that perfect speed-up is nearly impossible to achieve is that almost all programs have some sequential component, often the initialization phase, reading in the data, or collecting the results. Having many CPUs does not help here. Suppose that a program runs for T sec on a uniprocessor, with a fraction f of this time being sequential code and a fraction $(1 - f)$ being potentially parallelizable, as shown in Fig. 8-50(a). If the latter code can be run on n CPUs with no overhead, its execution time can be reduced from $(1 - f)T$ to $(1 - f)T/n$ at best, as shown in Fig. 8-50(b). This gives a total execution time for the sequential and parallel parts of $fT + (1 - f)T/n$. The speed-up is just the execution time of the original program, T, divided by this new execution time:

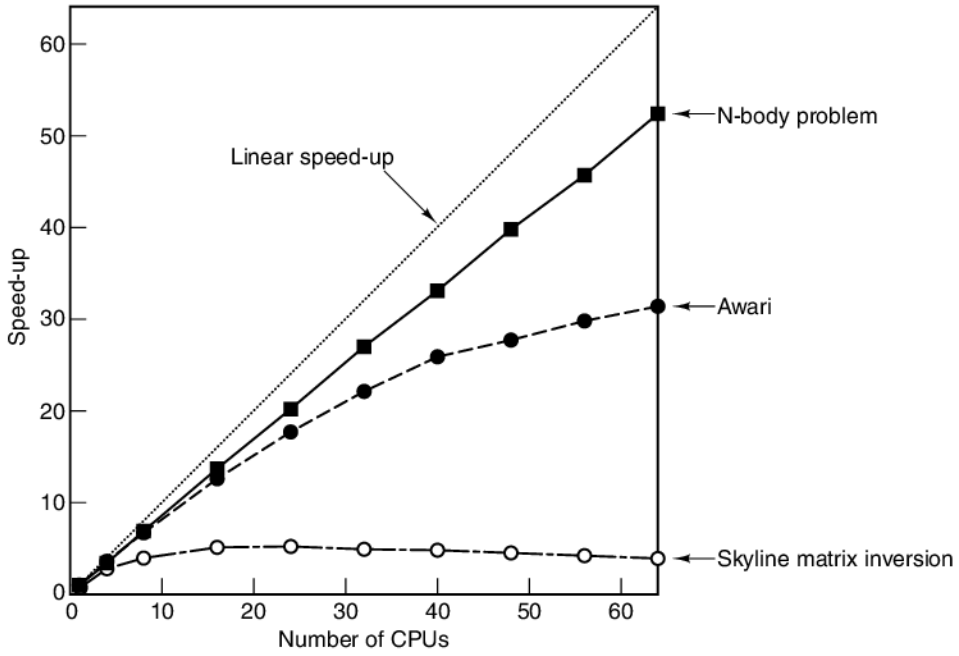

Figure 8-49. Real programs achieve less than linear speed-up.

$$Speed-up = \frac{n}{1 + (n-1)f}$$

For $f = 0$ we can get linear speed-up, but for $f > 0$, perfect speed-up is not possible due to the sequential component. This result is known as **Amdahl's law**.

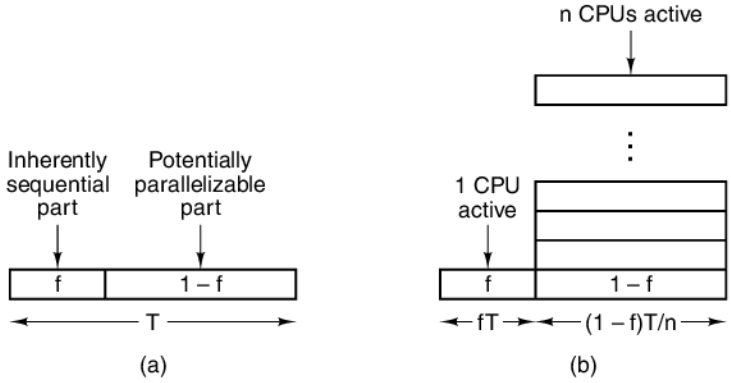

Figure 8-50. (a) A program has a sequential part and a parallelizable part. (b) Effect of running part of the program in parallel.

Amdahl's law is not the only reason perfect speed-up is nearly impossible to achieve. Nonzero communication latencies, finite communication bandwidths, and algorithmic inefficiencies can also play a role. Also, even if 1000 CPUs were available, not all programs can be written to make use of so many CPUs, and the overhead in getting them all started may be significant. Furthermore, sometimes the best-known algorithm does not parallelize well, so a suboptimal algorithm must be used in the parallel case. This all said, for many applications, having the program run n times faster is highly desirable, even if it takes $2n$ CPUs to do it. CPUs are not that expensive, after all, and many companies live with considerably less than 100% efficiency in other parts of their businesses.

Achieving High Performance

The most straightforward way to improve performance is to add more CPUs to the system. However, this addition must be done in such a way as to avoid creating any bottlenecks. A system in which one can add more CPUs and get correspondingly more computing power is said to be **scalable**.

To see some of the implications of scalability, consider four CPUs connected by a bus, as illustrated in Fig. 8-51(a). Now imagine scaling the system to 16 CPUs by adding 12 more, as shown in Fig. 8-51(b). If the bandwidth of the bus is b MB/sec, then by quadrupling the number of CPUs, we have also reduced the available bandwidth per CPU from b/4 MB/sec to b/16 MB/sec. Such a system is not scalable.

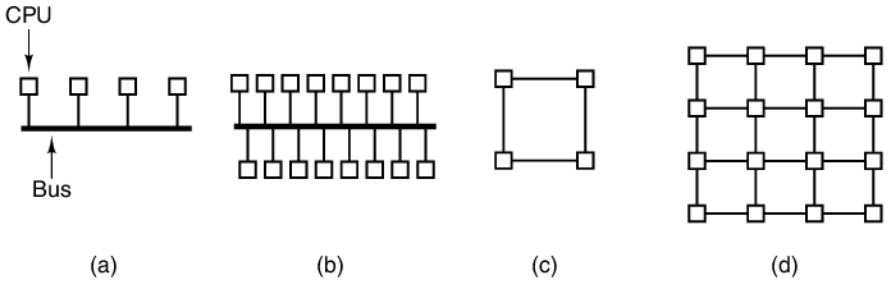

Figure 8-51. (a) A 4-CPU bus-based system. (b) A 16-CPU bus-based system. (c) A 4-CPU grid-based system. (d) A 16-CPU grid-based system.

Now we do the same thing with a grid-based system, as shown in Fig. 8-51(c) and Fig. 8-51(d). With this topology, adding new CPUs also adds new links, so scaling the system up does not cause the aggregate bandwidth per CPU to drop, as it does with a bus. In fact, the ratio of links to CPUs increases from 1.0 with 4 CPUs (4 CPUs, 4 links) to 1.5 with 16 CPUs (16 CPUs, 24 links), so adding CPUs improves the aggregate bandwidth per CPU.

Of course, bandwidth is not the only issue. Adding CPUs to the bus does not increase the diameter of the interconnection network or latency in the absence of traffic, whereas adding them to the grid does. For an $n \times n$ grid, the diameter is $2(n - 1)$, so the worst (and average) case latency increases roughly as the square root of the number of CPUs. For 400 CPUs, the diameter is 38, whereas for 1600 CPUs it is 78, so quadrupling the number of CPUs approximately doubles the diameter and thus the average latency.

Ideally, a scalable system should maintain the same average bandwidth per CPU and a constant average latency as more and more CPUs are added. In practice, however, keeping enough bandwidth per CPU is doable, but in all practical designs, latency grows with size. Having it grow logarithmically, as in a hypercube, is about the best that can be done.

The problem with having latency grow as the system scales up is that latency is often fatal to performance in fine- and medium-grained applications. If a program needs data that are not in its local memory, there is often a substantial delay in getting them, and the bigger the system, the longer the delay, as we have just seen. This problem is equally true of multiprocessors as of multicomputers, since in both cases the physical memory is invariably divided up into far-flung modules.

As a consequence of this observation, system designers often go to great lengths to reduce, or at least hide, the latency, using several techniques we will now mention. The first latency-hiding technique is data replication. If copies of a block of data can be kept at multiple locations, accesses from those locations can be speeded up. One such replication technique is caching, in which one or more copies of data blocks are kept close to where they are being used, as well as where they "belong." However, another strategy is to maintain multiple peer copies—copies that have equal status—as opposed to the asymmetric primary/secondary relationship used in caching. When multiple copies are maintained, in whatever form, key issues are where the data blocks are placed, when, and by whom. Answers range from dynamic placement on demand by the hardware, to intentional placement at load time following compiler directives. In all cases, managing consistency is an issue.

A second technique for hiding latency is **prefetching**. If a data item can be fetched before it is needed, the fetching process can be overlapped with normal execution, so that when the item is needed, it will be there. Prefetching can be automatic or under program control. When a cache loads not only the word being referenced, but an entire cache line containing the word, it is gambling that the succeeding words are also likely to be needed soon.

Prefetching can also be controlled explicitly. When the compiler realizes that it will need some data, it can put in an explicit instruction to go get them, and put that instruction sufficiently far in advance that the data will be there in time. This strategy requires that the compiler has a complete knowledge of the underlying machine and its timing, as well as control over where all data are placed. Such speculative LOAD instructions work best when it is known for sure that the data will

be needed. Getting a page fault on a LOAD for a path that is ultimately not taken is very costly.

A third technique that can hide latency is multithreading, as we have seen. If switching between processes can be made fast enough, for example, by giving each one its own memory map and hardware registers, then when one thread blocks waiting for remote data to arrive, the hardware can quickly switch to another one that is able to continue. In the limiting case, the CPU runs the first instruction from thread one, the second instruction from thread two, and so on. In this way, the CPU can be kept busy, even in the face of long memory latencies for the individual threads.

A fourth technique for hiding latency is using nonblocking writes. Normally, when a STORE instruction is executed, the CPU waits until the STORE has completed before continuing. With nonblocking writes, the memory operation is started, but the program just continues anyway. Continuing past a LOAD is harder, but with out-of-order execution, even that is possible.

8.5 GRID COMPUTING

Many of today's challenges in science, engineering, industry, the environment, and other areas are large scale and interdisciplinary. Solving them requires expertise, skills, knowledge, facilities, software, and data from multiple organizations, often in different countries. Some examples are as follows:

1. Scientists developing a mission to Mars.
2. A consortium building a complex product (e.g., a dam or aircraft).
3. An international relief team coordinating aid after a natural disaster.

Some of these are long-term cooperations, others are more short term, but they all share the common thread of requiring separate organizations with their own resources and procedures to work together to achieve a common goal.

Until recently, having different organizations, with different computers, operating systems, databases, and protocols work together to share resources and data has been very difficult. However, the growing need for large-scale interorganizational cooperation has led to the development of systems and technology for connecting widely separated computers into what is called the **grid**. In a sense, the grid is the next step along the axis of Fig. 8-1. It can be thought of as a very large, international, loosely coupled, heterogeneous, cluster.

The goal of the grid is to provide a technical infrastructure to allow a group of organizations that share a common goal to form a **virtual organization**. This virtual organization has to be flexible, with a large and changing membership, permitting the members to work together in areas they deem appropriate, while allowing them to maintain control over their own resources to whatever degree they wish.

To this end, grid researchers are developing services, tools, and protocols to enable these virtual organizations to function.

The grid is inherently multilateral with many participants who are peers. It can be contrasted with existing computing frameworks. In the client-server model, a transaction involves two parties: the server, who offers some service, and the client, who wants to use the service. A typical example of the client-server model is the Web, in which users go to Web servers to find information. The grid also differs from peer-to-peer applications, in which pairs of individuals exchange files. Email is a common example of a peer-to-peer application. Because the grid is different from these models, it requires new protocols and technology.

The grid needs access to a wide variety of resources. Each resource has a specific system and organization that owns it and that decides how much of the resource to make available to the grid, during which hours, and to whom. In an abstract sense, what the grid is about is resource access and management.

One way to model the grid is the layered hierarchy of Fig. 8-52. The **fabric layer** at the bottom is the set of components from which the grid is built. It includes CPUs, disks, networks, and sensors on the hardware side, and programs and data on the software side. These are the resources that the grid makes available in a controlled way.

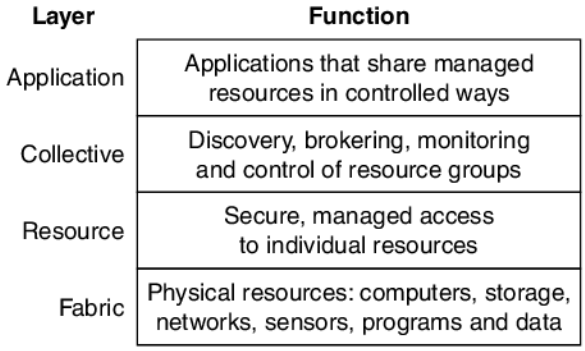

Figure 8-52. The grid layers.

One level higher is the **resource layer**, which manages the individual resources. In many cases, a resource participating in the grid has a local process that manages it and allows controlled access to it by remote users. This layer provides a uniform interface to higher layers for inquiring about the characteristics and status of individual resources, monitoring them, and using them in a secure way.

Next is the **collective layer**, which handles groups of resources. One of its functions is resource discovery, by which a user can locate available CPU cycles, disk space, or specific data. The collective layer may maintain directories or other databases to provide this information. It may also offer a brokering service by which the providers and users of services are matched up, possibly allocating scarce resources among competing users. The collective layer is also responsible

for replicating data, managing the admission of new members and resources to the grid, accounting, and maintaining the policy databases of who can use what.

Still further up is the **application layer**, where the user applications reside. It uses the lower layers to acquire credentials proving its right to use certain resources, submit usage requests, monitor the progress of these requests, deal with failures, and notify the user of the results.

Security is the key to a successful grid. Resource owners nearly always insist on maintaining tight control of their resources and want to determine who gets to use them, for how long, and how much. Without good security, no organization would make its resources available to the grid. On the other hand, if a user had to have a login account and password on every computer he wanted to use, using the grid would be unbearably cumbersome. Consequently, the grid has had to develop a security model to handle these concerns.

A key characteristic of the security model is the single sign-on. The first step in using the grid is to be authenticated and acquire a credential, a digitally signed document specifying on whose behalf the work is to be done. Credentials can be delegated, so that when a computation needs to create subcomputations, the child processes can also be identified. When a credential is presented at a remote machine, it has to be mapped onto the local security mechanism. On UNIX systems, for example, users are identified by 16-bit user IDs, but other systems have other schemes. Finally, the grid needs mechanisms to allow access policies to be stated, maintained, and updated.

In order to provide interoperability between different organizations and machines, standards are needed, in terms both of the services offered and of the protocols used to access them. The grid community has created an organization, the Global Grid Forum, to manage the standardization process. It has come up with a framework called **OGSA** (**Open Grid Services Architecture**) for positioning the various standards it is developing. Wherever possible, the standards utilize existing standards, for example, using WSDL (Web Services Definition Language) for describing OGSA services. The services being standardized currently fall into eight broad categories as follows, but no doubt new ones will be created later.

1. Infrastructure services (enable communication between resources).
2. Resource management services (reserve and deploy resources).
3. Data services (move and replicate data to where it is needed).
4. Context services (describe required resources and usage policies).
5. Information services (get information about resource availability).
6. Self-management services (support a stated quality of service).
7. Security services (enforce security policies).
8. Execution management services (manage workflow).

Much more could be said about the grid, but space limitations prevent us from pursuing this topic further. For more information about the grid, see Abramson (2011), Balasangameshwara and Raju (2012), Celaya and Arronategui (2011), Foster and Kesselman (2003), and Lee et al. (2011).

8.6 SUMMARY

It is getting increasingly difficult to make computers go faster by just revving up the clock due to increased heat dissipation problems and other factors. Instead, designers are looking to parallelism for speed-up. Parallelism can be introduced at many different levels, from very low, where the processing elements are very tightly coupled, to very high, where they are very loosely coupled.

At the bottom level is on-chip parallelism, in which parallel activities occur on a single chip. One form of on-chip parallelism is instruction-level parallelism, in which one instruction or a sequence of instructions issues multiple operations that can be executed in parallel by different functional units. A second form of on-chip parallelism is multithreading, in which the CPU can switch back and forth among multiple threads on an instruction-by-instruction basis, creating a virtual multiprocessor. A third form of on-chip parallelism is the single-chip multiprocessor, in which two or more cores are placed on the same chip to allow them to run at the same time.

One level up we find the coprocessors, typically plug-in boards that add extra processing power in some specialized area such as network protocol processing or multimedia. These extra processors relieve the main CPU of work, allowing it to do other things while they are performing their specialized tasks.

At the next level, we find the shared-memory multiprocessors. These systems contain two or more full-blown CPUs that share a common memory. UMA multiprocessors communicate via a shared (snooping) bus, a crossbar switch, or a multistage switching network. They are characterized by having a uniform access time to all memory locations. In contrast, NUMA multiprocessors also present all processes with the same shared address space, but here remote accesses take appreciably longer than local ones. Finally, COMA multiprocessors are yet another variation, in which cache lines move around the machine on demand but have no real home as in the other designs.

Multicomputers are systems with many CPUs that do not share a common memory. Each has its own private memory, with communication by message passing. MPPs are large multicomputers with specialized communication networks such as IBM's BlueGene/L. Clusters are simpler systems using off-the-shelf components, such as the engine that powers Google.

Multicomputers are often programmed using a message-passing package such as MPI. An alternative approach is to use application-level shared memory such as a page-based DSM system, the Linda tuple space, or Orca or Globe objects. DSM

simulates shared memory at the page level, making it similar to a NUMA machine, except with a much greater penalty for remote references.

Finally, at the highest level, and the most loosely coupled, are the grids. These are systems in which entire organizations are hooked together over the Internet to share compute power, data, and other resources.

PROBLEMS

1. Intel x86 instructions can be as long as 17 bytes. Is the x86 a VLIW CPU?
2. As process-design technology allows engineers to put ever more transistors on a chip, Intel and AMD have chosen to increase the number of cores on each chip. Are there any other feasible choices they could have made instead?
3. What are the clipped values of 96, −9, 300, and 256 when the clipping range is 0–255?
4. Are the following TriMedia instructions allowed, and if not, why not?

   a. Integer add, integer subtract, load, floating add, load immediate
   b. Integer subtract, integer multiply, load immediate, shift, shift
   c. Load immediate, floating add, floating multiply, branch, load immediate

5. Figure 8-7(d) and (e) show 12 cycles of instructions. For each one, tell what happens in the following three cycles.
6. On a particular CPU, an instruction that misses the level 1 cache but hits the level 2 cache takes k cycles in total. If multithreading is used to mask level 1 cache misses, how many threads must be run at once using fine-grained multithreading to avoid dead cycles?
7. The NVIDIA Fermi GPU is similar in spirit to one of the architectures we studied in Chap. 2. Which one?
8. One morning, the queen bee of a certain beehive calls in all her worker bees and tells them that today's assignment is to collect marigold nectar. The workers then fly off in different directions looking for marigolds. Is this an SIMD or an MIMD system?
9. During our discussion of memory consistency models, we said that a consistency model is a kind of contract between the software and the memory. Why is such a contract needed?
10. Consider a multiprocessor using a shared bus. What happens if two processors try to access the global memory at exactly the same instant?
11. Consider a multiprocessor using a shared bus. What happens if three processors try to access the global memory at exactly the same instant?
12. Suppose that for technical reasons it is possible for a snooping cache to snoop only on address lines, not on data lines. Would this change affect the write through protocol?

13. As a simple model of a bus-based multiprocessor system without caching, suppose that one instruction in every four references memory, and that a memory reference occupies the bus for an entire instruction time. If the bus is busy, the requesting CPU is put into a FIFO queue. How much faster will a 64-CPU system run than a 1-CPU system?

14. The MESI cache coherence protocol has four states. Other write-back cache coherence protocols have only three states. Which of the four MESI states could be sacrificed, and what would the consequences of each choice be? If you had to pick only three states, which would you pick?

15. Are there any situations with the MESI cache coherence protocol in which a cache line is present in the local cache but for which a bus transaction is nevertheless needed? If so, explain.

16. Suppose that there are n CPUs on a common bus. The probability that any CPU tries to use the bus in a given cycle is p. What is the chance that

a. The bus is idle (0 requests).
b. Exactly one request is made.
c. More than one request is made.

17. Name the major advantage and the major disadvantage of a crossbar switch.

18. How many crossbar switches does a full Sun Fire E25K have?

19. Suppose that the wire between switch 2A and switch 3B in the omega network of Fig. 8-31 breaks. Who is cut off from whom?

20. Hot spots (heavily referenced memory locations) are clearly a major problem in multistage switching networks. Are they also a problem in bus-based systems?

21. An omega switching network connects 4096 RISC CPUs, each with a 60-nsec cycle time, to 4096 infinitely fast memory modules. The switching elements each have a 5-nsec delay. How many delay slots are needed by a LOAD instruction?

22. Consider a machine using an omega switching network, like the one shown in Fig. 8-31. Suppose that the program and stack for processor i are kept in memory module i. Propose a slight change in the topology that makes a large difference in the performance (the IBM RP3 and BBN Butterfly use this modified topology). What disadvantage does your new topology have compared to the original?

23. In a NUMA multiprocessor, local memory references take 20 nsec and remote references 120 nsec. A certain program makes a total of N memory references during its execution, of which 1 percent are to a page P. That page is initially remote, and it takes C nsec to copy it locally. Under what conditions should the page be copied locally in the absence of significant use by other processors?

24. Consider a CC-NUMA multiprocessor like that of Fig. 8-33 except with 512 nodes of 8 MB each. If the cache lines are 64 bytes, what is the percentage overhead for the directories? Does increasing the number of nodes increase the overhead, decrease the overhead, or leave it unchanged?

25. What is the difference between NC-NUMA and CC-NUMA?

26. For each topology shown in Fig. 8-37, compute the diameter of the network.

27. For each topology shown in Fig. 8-37, determine the degree of fault tolerance each one has, defined as the maximum number of links that can be lost without partitioning the network in two.

28. Consider the double-torus topology of Fig. 8-37(f) but expanded to a size of $k \times k$. What is the diameter of the network? (*Hint*: Consider odd k and even k separately).

29. An interconnection network is in the form of an $8 \times 8 \times 8$ cube. Each link has a full-duplex bandwidth of 1 GB/sec. What is the bisection bandwidth of the network?

30. Amdahl's law limits the potential speed-up achievable on a parallel computer. Compute, as a function of f, the maximum possible speed-up as the number of CPUs approaches infinity. What are the implications of this limit for $f = 0.1$?

31. Figure 8-51 shows how scaling fails with a bus but succeeds with a grid. Assuming that each bus or link has a bandwidth b, compute the average bandwidth per CPU for each of the four cases. Then scale each system to 64 CPUs and repeat the calculations. What is the limit as the number of CPUs goes to infinity?

32. In the text, three variations of `send` were discussed: synchronous, blocking, and nonblocking. Give a fourth method that is similar to a blocking `send` but has slightly different properties. Give an advantage and a disadvantage of your method as compared to blocking `send`.

33. Consider a multicomputer running on a network with hardware broadcasting, such as Ethernet. Why does the ratio of read operations (those not updating internal state variables) to write operations (those updating internal state variables) matter?

9

BIBLIOGRAPHY

This chapter is an alphabetical bibliography of all books and articles cited in this book.

ABRAMSON, D.: "Mixing Cloud and Grid Resources for Many Task Computing," *Proc. Int'l Workshop on Many Task Computing on Grids and Supercomputers*, ACM, pp. 1–2, 2011.

ADAMS, M., and DULCHINOS, D.: "OpenCable," *IEEE Commun. Magazine*, vol. 39, pp. 98–105, June 2001.

ADIGA, N.R. et al.: "An Overview of the BlueGene/L Supercomputer," *Proc. Supercomputing 2002*, ACM, pp. 1–22, 2002.

ADVE, S.V., and HILL, M.: "Weak Ordering: A New Definition," *Proc. 17th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 2–14, 1990.

AGERWALA, T., and COCKE, J.: "High Performance Reduced Instruction Set Processors," IBM T.J. Watson Research Center Technical Report RC12434, 1987.

AHMADINIA, A., and SHAHRABI, A.: "A Highly Adaptive and Efficient Router Architecture for Network-on-Chip," *Computer J.*, vol. 54, pp. 1295–1307, Aug. 2011.

ALAM, S., BARRETT, R., BAST, M., FAHEY, M.R., KUEHN, J., McCURDY, ROGERS, J., ROTH, P., SANKARAN, R., VETTER, J.S., WORLEY, P., and YU, W.: "Early Evaluation of IBM BlueGene/P," *Proc. ACM/IEEE Conf. on Supercomputing*, ACM/IEEE, 2008.

ALAMELDEEN, A.R., and WOOD, D.A.: "Adaptive Cache Compression for High-Performance Processors," *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 212–223, 2004.

ALMASI, G.S. et al.: "System Management in the BlueGene/L Supercomputer," *Proc. 17th Int'l Parallel and Distr. Proc. Symp.*, IEEE, 2003a.

ALMASI, G.S. et al.: "An Overview of the Bluegene/L System Software Organization," *Par. Proc. Letters*, vol. 13, 561–574, April 2003b.

AMZA, C., COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., and ZWAENEPOEL, W.: "TreadMarks: Shared Memory Computing on a Network of Workstations," *IEEE Computer Magazine*, vol. 29, pp. 18–28, Feb. 1996.

ANDERSON, D.: *Universal Serial Bus System Architecture*, Reading, MA: Addison-Wesley, 1997.

ANDERSON, D., BUDRUK, R., and SHANLEY, T.: *PCI Express System Architecture*, Reading, MA: Addison-Wesley, 2004.

ANDERSON, T.E., CULLER, D.E., PATTERSON, D.A., and the NOW team: "A Case for NOW (Networks of Workstations)," *IEEE Micro Magazine*, vol. 15, pp. 54–64, Jan. 1995.

AUGUST, D.I., CONNORS, D.A., MAHLKE, S.A., SIAS, J.W., CROZIER, K.M., CHENG, B.-C., EATON, P.R., OLANIRAN, Q.B., and HWU, W.-M.: "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," *Proc. 25th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 227–237, 1998.

BAL, H.E.: *Programming Distributed Systems*, Hemel Hempstead, England: Prentice Hall Int'l, 1991.

BAL, H.E., BHOEDJANG, R., HOFMAN, R, JACOBS, C., LANGENDOEN, K., RUHL, T., and KAASHOEK, M.F.: "Performance Evaluation of the Orca Shared Object System," *ACM Trans. on Computer Systems*, vol. 16, pp. 1–40, Jan.–Feb. 1998.

BAL, H.E., KAASHOEK, M.F., and TANENBAUM, A.S.: "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Trans. on Software Engineering*, vol. 18, pp. 190–205, March 1992.

BAL, H.E., and TANENBAUM, A.S.: "Distributed Programming with Shared Data," *Proc. 1988 Int'l Conf. on Computer Languages*, IEEE, pp. 82–91, 1988.

BALASANGAMESHWARA, J., and RAJU, N.: "A Hybrid Policy for Fault Tolerant Load Balancing in Grid Computing Environments," *J. Network and Computer Applications*, vol. 35, pp. 412–422, Jan. 2012.

BARROSO, L.A., DEAN, J., and HOLZLE, U.: "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro Magazine*, vol. 23, pp. 22–28, March–April 2003.

BECHINI, A., CONTE, T.M., and PRETE, C.A.: "Opportunities and Challenges in Embedded Systems," *IEEE Micro Magazine*, vol. 24, pp. 8–9, July–Aug. 2004.

BHAKTHAVATCHALU, R., DEEPTHY, G.R.; and SHANOOJA, S.: "Implementation of Reconfigurable Open Core Protocol Compliant Memory System Using VHDL," *Proc. Int'l Conf. on Industrial and Information Systems*, pp. 213–218, 2010.

BJORNSON, R.D.: "Linda on Distributed Memory Multiprocessors," Ph.D. Thesis, Yale Univ., 1993.

BLUMRICH, M., CHEN, D., CHIU, G., COTEUS, P., GARA, A., GIAMPAPA, M.E., HARING, R.A., HEIDELBERGER, P., HOENICKE, D., KOPCSAY, G.V., OHMACHT, M., STEINMACHER-BUROW, B.D., TAKKEN, T., VRANSAS, P., and LIEBSCH, T.: "An Overview of the BlueGene/L System," *IBM J. Research and Devel.*, vol. 49, March–May, 2005.

BOSE, P.: "Computer Architecture Research: Shifting Priorities and Newer Challenges," *IEEE Micro Magazine*, vol. 24, p. 5, Nov.–Dec. 2004.

BOUKNIGHT, W.J., DENENBERG, S.A., MCINTYRE, D.E., RANDALL, J.M., SAMEH, A.H., and SLOTNICK, D.L.: "The Illiac IV System," *Proc. IEEE*, pp. 369–388, April 1972.

BRADLEY, D.: "A Personal History of the IBM PC," *IEEE Computer*, vol. 44, pp. 19–25, Aug. 2011.

BRIDE, E.: "The IBM Personal Computer: A Software-Driven Market," *IEEE Computer*, vol. 44, pp. 34–39, Aug. 2011.

BRIGHTWELL, R., CAMP, W., COLE, B., DEBENEDICTIS, E., LELAND, R., TOMPKINS, H, and MACCABE, A.B.: "Architectural Specification for Massively Parallel Supercomputers: An Experience-and-Measurement-Based Approach," *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 1–46, 2005.

BRIGHTWELL, R., UNDERWOOD, K.D., VAUGHAN, C., and STEVENSON, J.: "Performance Evaluation of the Red Storm Dual-Core Upgrade," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 175–190, Feb. 2010.

BURKHARDT, H., FRANK, S., KNOBE, B., and ROTHNIE, J.: "Overview of the KSR-1 Computer System," Technical Report KSR-TR-9202001, Kendall Square Research Corp., Cambridge, MA, 1992.

CARRIERO, N., and GELERNTER, D.: "Linda in Context," *Commun. of the ACM*, vol. 32, pp. 444–458, April 1989.

CELAYA, J., and ARRONATEGUI, U.: "A Highly Scalable Decentralized Scheduler of Tasks with Deadlines," *Proc. 12th Int'l Conf. on Grid Computing*, IEEE/ACM, pp. 58–65, 2011.

CHARLESWORTH, A.: "The Sun Fireplane Interconnect," *IEEE Micro Magazine*, vol. 22, pp. 36–45, Jan.–Feb. 2002.

CHARLESWORTH, A.: "The Sun Fireplane Interconnect," *Proc. Conf. on High Perf. Networking and Computing*, ACM, 2001.

CHEN, L., DROPSHO, S., and ALBONESI, D.H.: "Dynamic Data Dependence Tracking and Its Application to Branch Prediction," *Proc. Ninth Int'l Symp. on High-Performance Computer Arch.*, IEEE, pp. 65–78, 2003.

CHENG, L., and CARTER, J.B.: "Extending CC-NUMA Systems to Support Write Update Optimizations," *Proc. 2008 ACM/IEEE Conf. on Supercomputing*, ACM/IEEE, 2008.

CHOU, Y., FAHS, B., and ABRAHAM, S.: " Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 76–77, 2004.

COHEN, D.: "On Holy Wars and a Plea for Peace," *IEEE Computer Magazine*, vol. 14, pp. 48–54, Oct. 1981.

CORBATO, F.J., and VYSSOTSKY, V.A.: "Introduction and Overview of the MULTICS System," *Proc. FJCC*, pp. 185–196, 1965.

DENNING, P.J.: "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323–333, May 1968.

DIJKSTRA, E.W.: "GOTO Statement Considered Harmful," *Commun. of the ACM*, vol. 11, pp. 147–148, March 1968a.

DIJKSTRA, E.W.: "Co-operating Sequential Processes," in *Programming Languages*, F. Genuys (ed.), New York: Academic Press, 1968b.

DONALDSON, G., and JONES, D.: "Cable Television Broadband Network Architectures," *IEEE Commun. Magazine*, vol. 39, pp. 122–126, June 2001.

DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.: "Memory Access Buffering in Multiprocessors," *Proc. 13th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 434–442, 1986.

DULONG, C.: "The IA-64 Architecture at Work," *IEEE Computer Magazine*, vol. 31, pp. 24–32, July 1998.

DUTTA-ROY, A.: "An Overview of Cable Modem Technology and Market Perspectives," *IEEE Commun. Magazine*, vol. 39, pp. 81–88, June 2001.

FAGGIN, F., HOFF, M.E., Jr., MAZOR, S., and SHIMA, M.: "The History of the 4004," *IEEE Micro Magazine*, vol. 16, pp. 10–20, Nov. 1996.

FALCON, A., STARK, J., RAMIREZ, A., LAI, K., and VALERO, M.: "Prophet/Critic Hybrid Branch Prediction," *Proc. 31st Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 250–261, 2004.

FISHER, J.A., and FREUDENBERGER, S.M.: "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proc. Fifth Int'l Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, ACM, pp. 85–95, 1992.

FLYNN, D.: "AMBA: Enabling Reusable On-Chip Designs," *IEEE Micro Magazine*, vol. 17, pp. 20–27, July 1997.

FLYNN, M.J.: "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Computers*, vol. C-21, pp. 948–960, Sept. 1972.

FOSTER, I., and KESSELMAN, C.: *The Grid 2: Blueprint for a New Computing Infrastructure*, San Francisco: Morgan Kaufman, 2003.

FOTHERINGHAM, J.: "Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435–436, Oct. 1961.

FREITAS, H.C., MADRUGA, F.L., ALVES, M., and NAVAUX, P.: "Design of Interleaved Multithreading for Network Processors on Chip," *Proc. Int'l Symp. on Circuits and Systems*, IEEE, 2009.

GASPAR, L., FISCHER, V., BERNARD, F., BOSSUET, L., and COTRET, P.: "HCrypt: A Novel Concept of Crypto-processor with Secured Key Management," *Int'l Conf. on Reconfigurable Computing and FPGAs*, 2010.

GAUR, J., CHAUDHURI, C., and SUBRAMONEY, S.: "Bypass and Insertion Algorithms for Exclusive Last-level Caches," *Proc. 38th Int'l Symp. on Computer Arch.*, ACM, 2011.

GEBHART, M., JOHNSON, D.R., TARJAN, D., KECKLER, S.W., DALLY, W.J., LINDHOLM, E., and SKADRON, K.: "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," *Proc. 38th Int'l Symp. on Computer Arch.* ACM, 2011.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R., and SUNDERRAM, V.: *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Parallel Computing*, Cambridge, MA: MIT Press, 1994.

GEPNER, P., GAMAYUNOV, V., and FRASER, D.L.: "The 2nd Generation Intel Core Processor. Architectural Features Supporting HPC," *Proc. 10th Int'l Symp. on Parallel and Dist. Computing*, pp. 17–24, 2011.

GERBER, R., and BINSTOCK, A.: *Programming with Hyper-Threading Technology*, Santa Clara, CA: Intel Press, 2004.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P.B., GUPTA, A., and HENNESSY, J.L.: "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l Symp. on Comp. Arch.*, ACM, pp. 15–26, 1990.

GHEMAWAT, S., GOBIOFF, H., and LEUNG, S.-T.: "The Google File System," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 29–43, 2003.

GOODMAN, J.R.: "Using Cache Memory to Reduce Processor Memory Traffic," *Proc. 10th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 124–131, 1983.

GOODMAN, J.R.: "Cache Consistency and Sequential Consistency," Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

GOTH, G.: "IBM PC Retrospective: There Was Enough Right to Make It Work," *IEEE Computer*, vol. 44, pp. 26–33, Aug. 2011.

GROPP, W., LUSK, E., and SKJELLUM, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*, Cambridge, MA: MIT Press, 1994.

GUPTA, N., MANDAL, S., MALAVE, J., MANDAL, A., and MAHAPATRA, R.N.: "A Hardware Scheduler for Real Time Multiprocessor System on Chip," *Proc. 23rd Int'l Conf. on VLSI Design,* IEEE, 2010.

GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., and FRANKE, H.: "Reducing Disk Power Consumption in Servers with DRPM," *IEEE Computer Magazine*, vol. 36, pp. 59–66, Dec. 2003.

HAGERSTEN, E., LANDIN, A., and HARIDI, S.: "DDM—A Cache-Only Memory Architecture," *IEEE Computer Magazine*, vol. 25, pp. 44–54, Sept. 1992.

HAGHIGHIZADEH, F., ATTARZADEH, H., and SHARIFKHANI, M.: "A Compact 8-Bit AES Crypto-processor," *Proc. Second. Int'l Conf. on Computer and Network Tech.*, IEEE, 2010.

HAMMING, R.W.: "Error Detecting and Error Correcting Codes," *Bell Syst. Tech. J.*, vol. 29, pp. 147–160, April 1950.

HENKEL, J., HU, X.S., and BHATTACHARYYA, S.S.: "Taking on the Embedded System Challenge," *IEEE Computer Magazine*, vol. 36, pp. 35–37, April 2003.

HENNESSY, J.L.: "VLSI Processor Architecture," *IEEE Trans. on Computers*, vol. C-33, pp. 1221–1246, Dec. 1984.

HERRERO, E., GONZALEZ, J., and CANAL, R.: "Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors," *Proc. 23rd Int'l Conf. on VLSI Design,* IEEE, 2010.

HOARE, C.A.R.: "Monitors: An Operating System Structuring Concept," *Commun. of the ACM*, vol. 17, pp. 549–557, Oct. 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Feb. 1975.

HWU, W.-M.: "Introduction to Predicated Execution," *IEEE Computer Magazine*, vol. 31, pp. 49–50, Jan. 1998.

JIMENEZ, D.A.: "Fast Path-Based Neural Branch Prediction," *Proc. 36th Int'l Symp. on Microarchitecture*, IEEE, pp. 243–252, 2003.

JOHNSON, K.L., KAASHOEK, M.F., and WALLACH, D.A.: "CRL: High-Performance All-Software Distributed Shared Memory," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 213–228, 1995.

KAPASI, U.J., RIXNER, S., DALLY, W.J., KHAILANY, B., AHN, J.H., MATTSON, P., and OWENS, J.D.: "Programmable Stream Processors," *IEEE Computer Magazine*, vol. 36, pp. 54–62, Aug. 2003.

KAUFMAN, C., PERLMAN, R., and SPECINER, M.: *Network Security,* 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2002.

KIM, N.S., AUSTIN, T., BLAAUW, D., MUDGE, T., FLAUTNER, K., HU, J.S., IRWIN, M.J., KANDEMIR, M., and NARAYANAN, V.: "Leakage Current: Moore's Law Meets Static Power," *IEEE Computer Magazine*, vol. 36, 68–75, Dec. 2003.

KNUTH, D.E.: *The Art of Computer Programming: Fundamental Algorithms,* 3rd ed., Reading, MA: Addison-Wesley, 1997.

KONTOTHANASSIS, L., HUNT, G., STETS, R., HARDAVELLAS, N., CIERNIAD, M., PARTHASARATHY, S., MEIRA, W., DWARKADAS, S., and SCOTT, M.: "VM-Based Shared Memory on Low Latency Remote Memory Access Networks," *Proc. 24th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 157–169, 1997.

LAMPORT, L.: "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, vol. C-28, pp. 690–691, Sept. 1979.

LAROWE, R.P., and ELLIS, C.S.: "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 319–363, Nov. 1991.

LEE, J., KELEHER, P., and SUSSMAN, A.: "Supporting Computing Element Heterogeneity in P2P Grids," *Proc. IEEE Int'l Conf. on Cluster Computing*, IEEE, pp. 150–158, 2011.

LI, K., and HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321–359, Nov. 1989.

LIN, Y.-N., LIN, Y.-D., and LAI, Y.-C.: "Thread Allocation in CMP-based Multithreaded Network Processors," *Parallel Computing*, vol. 36, pp. 104–116, Feb. 2010.

LU, H., COX, A.L., DWARKADAS, S., RAJAMONY, R., and ZWAENEPOEL, W.: "Software Distributed Shared Memory Support for Irregular Applications," *Proc. Sixth Conf. on Prin. and Practice of Parallel Progr.*, pp. 48–56, June 1997.

LUKASIEWICZ, J.: *Aristotle's Syllogistic,* 2nd ed., Oxford: Oxford University Press, 1958.

LYYTINEN, K., and YOO, Y.: "Issues and Challenges in Ubiquitous Computing," *Commun. of the ACM*, vol. 45, pp. 63–65, Dec. 2002.

MARTIN, R.P., VAHDAT, A.M., CULLER, D.E., and ANDERSON, T.E.: "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proc. 24th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 85–97, 1997.

MAYHEW, D., and KRISHNAN, V.: "PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects," *Proc. 11th Symp. on High Perf. Interconnects*, IEEE, pp. 21–29, Aug. 2003.

McKUSICK, M.K., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.: "A Fast File System for UNIX," *ACM Trans. on Computer Systems*, vol. 2, pp. 181–197, Aug. 1984.

McNAIRY, C., and SOLTIS, D.: "Itanium 2 Processor Microarchitecture," *IEEE Micro Magazine*, vol. 23, pp. 44–55, March-April 2003.

MISHRA, A.K., VIJAYKRISHNAN, N., and DAS, C.R.: "A Case for Heterogeneous On-Chip Interconnects for CMPs," *Proc. 38th Int'l Symp. on Computer Arch.* ACM, 2011.

MORGAN, C.: *Portraits in Computing*, New York: ACM Press, 1997.

MOUDGILL, M., and VASSILIADIS, S.: "Precise Interrupts," *IEEE Micro Magazine*, vol. 16, pp. 58–67, Jan. 1996.

MULLENDER, S.J., and TANENBAUM, A.S.: "Immediate Files," *Software—Practice and Experience*, vol. 14, pp. 365–368, 1984.

NAEEM, A., CHEN, X., LU, Z., and JANTSCH, A.: "Realization and Performance Comparison of Sequential and Weak Memory Consistency Models in Network-On-Chip Based Multicore Systems," *Proc. 16th Design Automation Conf. Asia and South Pacific*, IEEE, pp. 154–159, 2011.

ORGANICK, E.: *The MULTICS System*, Cambridge, MA: MIT Press, 1972.

OSKIN, M., CHONG, F.T., and CHUANG, I.L.: "A Practical Architecture for Reliable Quantum Computers," *IEEE Computer Magazine*, vol. 35, pp. 79–87, Jan. 2002.

PAPAMARCOS, M., and PATEL., J.: "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proc. 11th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 348–354, 1984.

PARIKH, D., SKADRON, K., ZHANG, Y., and STAN, M.: "Power-Aware Branch Prediction: Characterization and Design," *IEEE Trans. on Computers*, vol. 53, 168–186, Feb. 2004.

PATTERSON, D.A.: "Reduced Instruction Set Computers," *Commun. of the ACM*, vol. 28, pp. 8–21, Jan. 1985.

PATTERSON, D.A., GIBSON, G., and KATZ, R.: "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, ACM, pp. 109–166, 1988.

PATTERSON, D.A., and SEQUIN, C.H.: "A VLSI RISC," *IEEE Computer Magazine*, vol. 15, pp. 8–22, Sept. 1982.

POUNTAIN, D.: "Pentium: More RISC than CISC," *Byte*, vol. 18, pp. 195–204, Sept. 1993.

RADIN, G.: "The 801 Minicomputer," *Computer Arch. News*, vol. 10, pp. 39–47, March 1982.

RAMAN, S.K., PENTKOVSKI, V., and KESHAVA, J.: "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE Micro Magazine*, vol. 20, pp. 47–57, July-Aug. 2000.

RITCHIE, D.M.: "Reflections on Software Research," *Commun. pf the ACM*, vol. 27, pp. 758–760, Aug. 1984.

RITCHIE, D.M., and THOMPSON, K.: "The UNIX Time-Sharing System," *Commun. of the ACM*, vol. 17, pp. 365–375, July 1974.

ROBINSON, G.S.: "Toward the Age of Smarter Storage," *IEEE Computer Magazine*, vol. 35, pp. 35–41, Dec. 2002.

ROSENBLUM, M., and OUSTERHOUT, J.K.: "The Design and Implementation of a Log-Structured File System," *Proc. Thirteenth Symp. on Operating System Principles*, ACM, pp. 1–15, 1991.

RUSSINOVICH, M.E., and SOLOMON, D.A.: *Microsoft Windows Internals,* 4th ed., Redmond, WA: Microsoft Press, 2005.

RUSU, S., MULJONO, H., and CHERKAUER, B.: "Itanium 2 Processor 6M," *IEEE Micro Magazine*, vol. 24, pp. 10–18, March–April 2004.

SAHA, D., and MUKHERJEE, A.: "Pervasive Computing: A Paradigm for the 21st Century," *IEEE Computer Magazine*, vol. 36, pp. 25–31, March 2003.

SAKAMURA, K.: "Making Computers Invisible," *IEEE Micro Magazine*, vol. 22, pp. 7–11, 2002.

SANCHEZ, D., and KOZYRAKIS, C.: "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. 38th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 57–68, 2011.

SCALES, D.J., GHARACHORLOO, K., and THEKKATH, C.A.: "Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory," *Proc. Seventh Int'l Conf. on Arch. Support for Prog. Lang. and Oper. Syst.*, ACM, pp. 174–185, 1996.

SELTZER, M., BOSTIC, K., McKUSICK, M.K., and STAELIN, C.: "An Implementation of a Log-Structured File System for UNIX," *Proc. Winter 1993 USENIX Technical Conf.*, pp. 307–326, 1993.

SHANLEY, T., and ANDERSON, D.: *PCI System Architecture,* 4th ed., Reading, MA: Addison-Wesley, 1999.

SHOUFAN, A., HUBER, N., and MOLTER, H.G.: "A Novel Cryptoprocessor Architecture for Chained Merkle Signature Schemes," *Microprocessors and Microsystems*, vol. 35, pp. 34–47, Feb. 2011.

SINGH, G.: "The IBM PC: The Silicon Story," *IEEE Computer*, vol. 44, pp. 40–45, Aug. 2011.

SLATER, R.: *Portraits in Silicon*, Cambridge, MA: MIT Press, 1987.

SNIR, M., OTTO, S.W., HUSS-LEDERMAN, S., WALKER, D.W., and DONGARRA, J.: *MPI: The Complete Reference Manual*, Cambridge, MA: MIT Press, 1996.

SOLARI, E., and CONGDON, B.: *PCI Express Design & System Architecture*, Research Tech, Inc., 2005.

SOLARI, E., and WILLSE, G.: *PCI and PCI-X Hardware and Software,* 6th ed., San Diego, CA: Annabooks, 2004.

SORIN, D.J., HILL, M.D., and WOOD, D.A.: *A Primer on Memory Consistency and Cache Coherence*, San Francisco: Morgan & Claypool, 2011.

STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., and SCOTT, M.: "CASHMERE-2L: Software Coherent Shared Memory on Clustered Remote-Write Networks," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 170–183, 1997.

SUMMERS, C.K.: *ADSL: Standards, Implementation, and Architecture*, Boca Raton, FL: CRC Press, 1999.

SUNDERRAM, V.B..: "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, Dec. 1990.

SWAN, R.J., FULLER, S.H., and SIEWIOREK, D.P.: "Cm*—A Modular Multiprocessor," *Proc. NCC*, pp. 645–655, 1977.

TAN, W.M.: *Developing USB PC Peripherals*, San Diego, CA: Annabooks, 1997.

TANENBAUM, A.S., and WETHERALL, D.J.: *Computer Networks,* 5th ed., Upper Saddle River, NJ: Prentice Hall, 2011.

THOMPSON, K.: "Reflections on Trusting Trust," *Commun. of the ACM*, vol. 27, pp. 761–763, Aug. 1984.

THOMPSON, J., DREISIGMEYER, D.W., JONES, T., KIRBY, M., and LADD, J.: "Accurate Fault Prediction of BlueGene/P RAS Logs via Geometric Reduction," IEEE, pp. 8–14, 2010.

TRELEAVEN, P.: "Control-Driven, Data-Driven, and Demand-Driven Computer Architecture," *Parallel Computing*, vol. 2, 1985.

TU, X., FAN, X., JIN, H., ZHENG, L., and PENG, X.: "Transactional Memory Consistency: A New Consistency Model for Distributed Transactional Memory," *Proc. Third Int'l Joint Conf. on Computational Science and Optimization*, IEEE, 2010.

VAHALIA, U.: *UNIX Internals*, Upper Saddle River, NJ: Prentice Hall, 1996.

VAHID, F.: "The Softening of Hardware," *IEEE Computer Magazine*, vol. 36, pp. 27–34, April 2003.

VETTER, P., GODERIS, D., VERPOOTEN, L., and GRANGER, A.: "Systems Aspects of APON/VDSL Deployment," *IEEE Commun. Magazine*, vol. 38, pp. 66–72, May 2000.

VU, T.D., ZHANG, L., and JESSHOPE, C.: "The Verification of the On-Chip COMA Cache Coherence Protocol," *Proc. 12th Int'l Conf. on Algebraic Methodology and Software Technology*, Springer-Verlag, pp. 413–429, 2008.

WEISER, M.: "The Computer for the 21st Century," *IEEE Pervasive Computing*, vol. 1, pp. 19–25, Jan.–March 2002; originally published in *Scientific American*, Sept. 1991.

WILKES, M.V.: "Computers Then and Now," *J. ACM*, vol. 15, pp. 1–7, Jan. 1968.

WILKES, M.V.: "The Best Way to Design an Automatic Calculating Machine," *Proc. Manchester Univ. Computer Inaugural Conf.*, 1951.

WING-KEI, Y., HUANG, R., XU, S., WANG, S.-E., KAN, E., and SUH, G.E.: "SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading Architectures," *Proc. 38th Int'l Symp. on Computer Arch.* ACM, 2011.

YAMAMOTO, S., and NAKAO, A.: "Fast Path Performance of Packet Cache Router Using Multi-core Network Processor," *Proc. Seventh Symp. on Arch. for Network and Comm. Sys.*, ACM/IEEE, 2011.

ZHANG, L., and JESSHOPE, C.: "On-Chip COMA Cache-Coherence Protocol for Micro-grids of Microthreaded Cores," *Proc. of 2007 European Conf. on Parallel Processing*, Springer-Verlag, pp. 38–48, 2008.

A

BINARY NUMBERS

The arithmetic used by computers differs in some ways from the arithmetic used by people. The most important difference is that computers perform operations on numbers whose precision is finite and fixed. Another difference is that most computers use the binary rather than the decimal system for representing numbers. These topics are the subject of this appendix.

A.1 FINITE-PRECISION NUMBERS

While doing arithmetic, one usually gives little thought to the question of how many decimal digits it takes to represent a number. Physicists can calculate that there are 10^{78} electrons in the universe without being bothered by the fact that it requires 79 decimal digits to write that number out in full. Someone calculating the value of a function with pencil and paper who needs the answer to six significant digits simply keeps intermediate results to seven, or eight, or however many are needed. The problem of the paper not being wide enough for seven-digit numbers never arises.

With computers, matters are quite different. On most computers, the amount of memory available for storing a number is fixed at the time that the computer is designed. With a certain amount of effort, the programmer can represent numbers two, or three, or even many times larger than this fixed amount, but doing so does not change the nature of this difficulty. The finite nature of the computer forces us

to deal only with numbers that can be represented in a fixed number of digits. We call such numbers **finite-precision numbers**.

In order to study properties of finite-precision numbers, let us examine the set of positive integers representable by three decimal digits, with no decimal point and no sign. This set has exactly 1000 members: 000, 001, 002, 003, ..., 999. With this restriction, it is impossible to express certain kinds of numbers, such as

1. Numbers larger than 999.
2. Negative numbers.
3. Fractions.
4. Irrational numbers.
5. Complex numbers.

One important property of arithmetic on the set of all integers is **closure** with respect to the operations of addition, subtraction, and multiplication. In other words, for every pair of integers i and j, $i + j$, $i - j$, and $i \times j$ are also integers. The set of integers is not closed with respect to division, because there exist values of i and j for which i/j is not expressible as an integer (e.g., 7/2 and 1/0).

Finite-precision numbers are not closed with respect to any of these four basic operations, as shown below, using three-digit decimal numbers as an example:

600 + 600 = 1200	(too large)
003 − 005 = −2	(negative)
050 × 050 = 2500	(too large)
007 / 002 = 3.5	(not an integer)

The violations can be divided into two mutually exclusive classes: operations whose result is larger than the largest number in the set (overflow error) or smaller than the smallest number in the set (underflow error), and operations whose result is neither too large nor too small but is simply not a member of the set. Of the four violations above, the first three are examples of the former, and the fourth is an example of the latter.

Because computers have finite memories and therefore must of necessity perform arithmetic on finite-precision numbers, the results of certain calculations will be, from the point of view of classical mathematics, just plain wrong. A calculating device that gives the wrong answer even though it is in perfect working condition may appear strange at first, but the error is a logical consequence of its finite nature. Some computers have special hardware that detects overflow errors.

The algebra of finite-precision numbers is different from normal algebra. As an example, consider the associative law:

$$a + (b - c) = (a + b) - c$$

Let us evaluate both sides for $a = 700$, $b = 400$, $c = 300$. To compute the left-hand

side, first calculate $(b - c)$, which is 100, and then add this amount to a, yielding 800. To compute the right-hand side, first calculate $(a + b)$, which gives an overflow in the finite arithmetic of three-digit integers. The result may depend on the machine being used but it will not be 1100. Subtracting 300 from some number other than 1100 will not yield 800. The associative law does not hold. The order of operations is important.

As another example, consider the distributive law:

$$a \times (b - c) = a \times b - a \times c$$

Let us evaluate both sides for $a = 5$, $b = 210$, $c = 195$. The left-hand side is 5×15, which yields 75. The right-hand side is not 75 because $a \times b$ overflows.

Judging from these examples, one might conclude that although computers are general-purpose devices, their finite nature renders them especially unsuitable for doing arithmetic. This conclusion is, of course, not true, but it does serve to illustrate the importance of understanding how computers work and what limitations they have.

A.2 RADIX NUMBER SYSTEMS

An ordinary decimal number with which everyone is familiar consists of a string of decimal digits and, possibly, a decimal point. The general form and its usual interpretation are shown in Fig. A-1. The choice of 10 as the base for exponentiation, called the **radix**, is made because we are using decimal, or base 10, numbers. When dealing with computers, it is frequently convenient to use radices other than 10. The most important radices are 2, 8, and 16. The number systems based on these radices are called **binary**, **octal**, and **hexadecimal**, respectively.

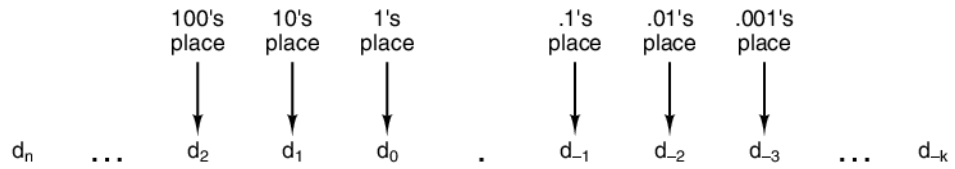

Figure A-1. The general form of a decimal number.

A radix k number system requires k different symbols to represent the digits 0 to $k - 1$. Decimal numbers are built up from the 10 decimal digits

0 1 2 3 4 5 6 7 8 9

In contrast, binary numbers do not use these ten digits. They are all constructed exclusively from the two binary digits

0 1

Octal numbers are built up from the eight octal digits

0 1 2 3 4 5 6 7

For hexadecimal numbers, 16 digits are needed. Thus six new symbols are required. It is conventional to use the uppercase letters A through F for the six digits following 9. Hexadecimal numbers are then built up from the digits

0 1 2 3 4 5 6 7 8 9 A B C D E F

The expression "binary digit" meaning a 1 or a 0 is usually referred to as a **bit**. Figure A-2 shows the decimal number 2001 expressed in binary, octal, decimal, and hexadecimal form. The number 7B9 is obviously hexadecimal, because the symbol B can only occur in hexadecimal numbers. However, the number 111 might be in any of the four number systems discussed. To avoid ambiguity, people use a subscript of 2, 8, 10, or 16 to indicate the radix when it is not obvious from the context.

Binary	1	1	1	1	1	0	1	0	0	0	1
	1×2^{10}	$+\ 1\times2^{9}$	$+\ 1\times2^{8}$	$+\ 1\times2^{7}$	$+\ 1\times2^{6}$	$+\ 0\times2^{5}$	$+\ 1\times2^{4}$	$+\ 0\times2^{3}$	$+\ 0\times2^{2}$	$+\ 0\times2^{1}$	$+\ 1\times2^{0}$
	1024	+ 512	+ 256	+ 128	+ 64	+ 0	+16	+ 0	+ 0	+ 0	+ 1
Octal	3	7	2	1							
	3×8^{3}	$+\ 7\times8^{2}$	$+\ 2\times8^{1}$	$+\ 1\times8^{0}$							
	1536	+ 448	+ 16	+ 1							
Decimal	2	0	0	1							
	2×10^{3}	$+\ 0\times10^{2}$	$+\ 0\times10^{1}$	$+\ 1\times10^{0}$							
	2000	+ 0	+ 0	+ 1							
Hexadecimal	7	D	1								
	7×16^{2}	$+\ 13\times16^{1}$	$+\ 1\times16^{0}$								
	1792	+ 208	+ 1								

Figure A-2. The number 2001 in binary, octal, and hexadecimal.

As an example of binary, octal, decimal, and hexadecimal notation, consider Fig. A-3, which shows a collection of nonnegative integers expressed in each of these four different systems. Perhaps some archaeologist thousands of years from now will discover this table and regard it as the Rosetta Stone to late twentieth century and early twenty-first century number systems.

Decimal	Binary	Octal	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	3	3
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Figure A-3. Decimal numbers and their binary, octal, and hexadecimal equivalents.

A.3 CONVERSION FROM ONE RADIX TO ANOTHER

Conversion between octal or hexadecimal numbers and binary numbers is easy. To convert a binary number to octal, divide it into groups of 3 bits, with the 3 bits immediately to the left (or right) of the decimal point (often called a binary point) forming one group, the 3 bits immediately to their left, another group, and so on. Each group of 3 bits can be directly converted to a single octal digit, 0 to 7, according to the conversion given in the first lines of Fig. A-3. It may be necessary to add one or two leading or trailing zeros to fill out a group to 3 full bits. Conversion from octal to binary is equally trivial. Each octal digit is simply replaced by the equivalent 3-bit binary number. Conversion from hexadecimal to binary is just like

octal-to-binary except that each hexadecimal digit corresponds to a group of 4 bits instead of 3 bits. Figure A-4 gives some examples of conversions.

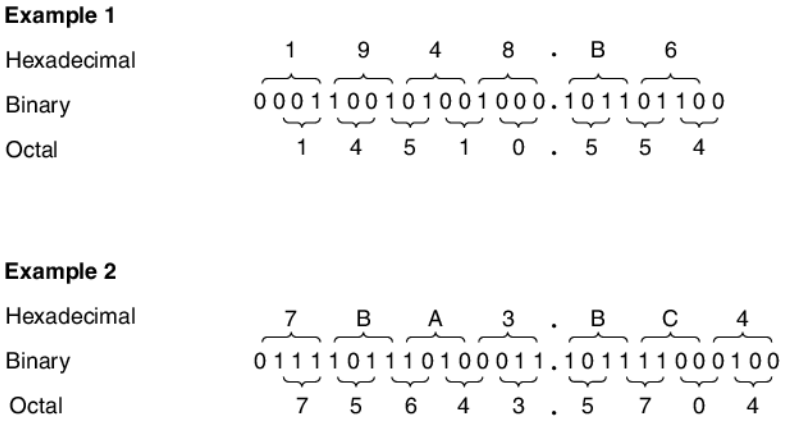

Figure A-4. Examples of octal-to-binary and hexadecimal-to-binary conversion.

Conversion of decimal numbers to binary can be done in two different ways. The first method follows directly from the definition of binary numbers. The largest power of 2 smaller than the number is subtracted from the number. The process is then repeated on the difference. Once the number has been decomposed into powers of 2, the binary number can be assembled with 1s in the bit positions corresponding to powers of 2 used in the decomposition, and 0s elsewhere.

The other method (for integers only) consists of dividing the number by 2. The quotient is written directly beneath the original number and the remainder, 0 or 1, is written next to the quotient. The quotient is then considered and the process repeated until the number 0 has been reached. The result of this process will be two columns of numbers, the quotients and the remainders. The binary number can now be read directly from the remainder column starting at the bottom. Figure A-5 gives an example of decimal-to-binary conversion.

Binary integers can also be converted to decimal in two ways. One method consists of summing up the powers of 2 corresponding to the 1 bits in the number. For example,

$$10110 = 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$$

In the other method, the binary number is written vertically, one bit per line, with the leftmost bit on the bottom. The bottom line is called line 1, the one above it line 2, and so on. The decimal number will be built up in a parallel column next to the binary number. Begin by writing a 1 on line 1. The entry on line n consists of two times the entry on line $n - 1$ plus the bit on line n (either 0 or 1). The entry on the top line is the answer. Figure A-6 gives an example of this method of binary to decimal conversion.

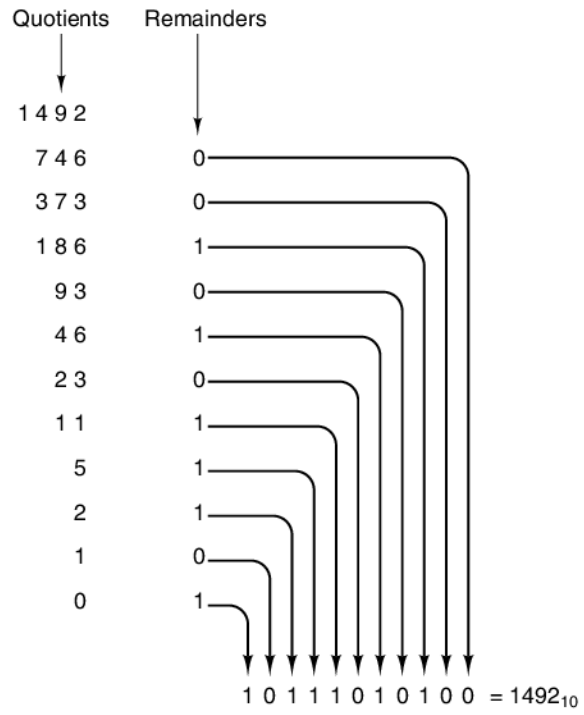

Figure A-5. Conversion of the decimal number 1492 to binary by successive halving, starting at the top and working downward. For example, 93 divided by 2 yields a quotient of 46 and a remainder of 1, written on the line below it.

Decimal-to-octal and decimal-to-hexadecimal conversion can be accomplished either by first converting to binary and then to the desired system or by subtracting powers of 8 or 16.

A.4 NEGATIVE BINARY NUMBERS

Four different systems for representing negative numbers have been used in digital computers at one time or another in history. The first one is called **signed magnitude**. In this system the leftmost bit is the sign bit (0 is + and 1 is −) and the remaining bits hold the absolute magnitude of the number.

The second system, called **one's complement**, also has a sign bit with 0 used for plus and 1 for minus. To negate a number, replace each 1 by a 0 and each 0 by a 1. This holds for the sign bit as well. One's complement is obsolete.

The third system, called **two's complement**, also has a sign bit that is 0 for plus and 1 for minus. Negating a number is a two-step process. First, each 1 is replaced by a 0 and each 0 by a 1, just as in one's complement. Second, 1 is added