



Computer Architecture

Copyright (C) 1991, 1992, 1993, 1994, 1995 by the Computational Science Education Project

This electronic book is copyrighted, and protected by the copyright laws of the United States. This (and all associated documents in the system) must contain the above copyright notice. If this electronic book is used anywhere other than the project's original system, CSEP must be notified in writing (email is acceptable) and the copyright notice must remain intact.

1 Overview

The advantages of programming in a high level language include abstraction and portability. Abstraction means programmers can describe algorithms in a "high level" notation that is independent of details about the machine that will execute the algorithm. Portability is a byproduct of abstraction that allows programs to be run on a wide variety of computers as long as there is a compiler that will translate them for each machine.

In most programming situations reality is close to the ideal. Compilers for many high level languages are very good at generating efficient and portable code for typical computer systems, so programmers are able to express algorithms in high level languages and expect them to run efficiently on almost any machine. There may be a few isolated places where a programmer who invests a lot of effort may be able to write a more efficient routine in assembly language (the native language of the machine), but it is hardly ever worth the effort to write an entire program in assembly language. Obviously when all or part of a program is written in assembler it is not as abstract, since assembly language is the language of the machine and not the language of the application, and it is no longer portable from one machine to another.

It is not necessary to write a lower level program in order to compromise abstraction and portability. As a simple example, suppose a program multiplies a value x by 2. The obvious way to write this in C is " $2*x$ ". But if a programmer knows the machine that will execute this program has a slow multiplication instruction, and knows that integers are represented in binary, she can use the expression " $x \ll 1$ " instead.¹ The resulting program is less abstract

¹In C " $2*x$ " means "shift x left one bit", an operation that most machines do very efficiently. The binary number system and why shifting left is equivalent to multiplication will be described later in this chapter, and the C programming language is described in the chapter on programming languages.

efficient as they could be and to transform them so they make better use of the underlying machine. We cannot hope to present a comprehensive collection of performance tips for languages and systems likely to be used in computational science. Rather we aim to give enough background information on common structures such as vector processors and cache memories so you will be able to (a) recognize when your program is not performing near the capacity of your system, (b) understand performance improvement techniques recommended by the compiler writers and/or system architects of your system, and (c) decide whether the benefits of increased performance are worth sacrificing abstraction and portability.

Another, closely related, goal is to provide the necessary background in computer architecture to evaluate competing algorithms to decide which is likely to be the most efficient for a given machine, even before they are expressed in a programming language. Performance will depend on several factors, for example how data is laid out in memory and the patterns in which it is accessed. In many cases an algorithm with worse asymptotic complexity may turn out to be the best for a certain class of machines because it requires fewer processors or less memory (see Numerical Linear Algebra).

The chapter begins with an overview of basic computer architecture. It describes the main components of a typical system and how they interact. Section 3, is on the architecture and implementation of modern high performance systems, including vector processors and parallel processors; readers who are already familiar with basic computer architecture may want to skip directly to this section. In both sections we will describe the basic concepts, introduce performance models, and discuss factors that limit efficient use of the machines. Programming issues, for example organizing loops so they can be executed more efficiently by a vector processor or accessing data structures in ways that are most efficient for certain memory systems, are discussed in the chapter on programming languages

2 Basic Computer Architecture

The main components in a typical computer system are the processor, memory, input/output devices, and the communication channels that connect them.

The processor is the workhorse of the system; it is the component that executes a program by performing arithmetic and logical operations on data. It is the only component that creates new information by combining or modifying current information. In a typical system there will be only one processor, known at the *central processing unit*, or CPU. Modern high performance systems, for example vector processors and parallel processors, often have more than one processor. Systems with only one processor are serial processors, or, especially among computational scientists, *scalar processors*.

Memory is a passive component that simply stores information until it is requested by another part of the system. During normal operations it feeds instructions and data to the processor, and at other times it is the source or destination of data transferred by I/O devices. Information in a memory is accessed by its *address*. In programming language terms, one can view memory as a one-dimensional array M . A processor's request to the memory might

than the original, since one of its operations is defined in machine level terms (shifting a pattern of bits) instead of mathematical terms (multiplication). It is less portable, since it now runs only on machines that use binary to represent integers (a pretty safe bet) and is efficient only on machines that shift bits in a single operation.

There are situations where programmers need to use knowledge of the underlying computer system in order to optimize programs written in high level languages, and computational scientists will often find themselves in these situations. If a program runs for days, or even weeks, an optimization that improves execution by just a few percent can save many hours, which translates into real savings if the program runs at a supercomputer center where the scientist pays for CPU time. Another factor is that high performance computers used by computational scientists are much more complicated than other machines, and a compiler may not be able to translate a program efficiently without a little help from the programmer. A common situation is a loop written in Fortran, which, if written carefully, can be translated into a single instruction for a vector processor. Computational scientists often use the newest machines, and these are the machines most likely to have immature compilers. It takes several years experience with real programs for compiler writers to learn how to develop optimizations that will more fully exploit the capabilities of the underlying machine, and in many cases the theory behind the optimizations has yet to be worked out. For example, techniques for optimal mapping of independent pieces of a parallel program so they can be executed simultaneously on different nodes in a parallel processor is an active area of research in computer science. Programmers who use parallel processors often need to allocate tasks themselves using system-dependent library routines to send information from one task to another. Knowing how processors are interconnected will have an impact on how efficiently messages are passed.

Computer scientists used three related terms to describe the general area of low-level machine organization. *Computer architecture* is the study of the components that make up computer systems and how they are interconnected. *Computer organization* is concerned with the implementation of a computer architecture. As an example of the difference between architecture and implementation, consider the vector supercomputers from Cray Research. These machines have very similar architectures from a programmer's point of view: processors in these systems have the same number of internal registers (temporary storage) for both vector and scalar data, they have the same basic instruction set, and operands in main memory have the same formats. The systems have very different organizations, however, since they may have a different number of processors, memory sizes may vary, operands are transferred from memory to the processor in different ways, and the time to execute an instruction varies from one system to another. *Computer engineering* refers to the actual construction of a system: lengths of wires, sizes of circuits, cooling and electrical requirements, etc. Programmers often use knowledge of a system's architecture, and sometimes organization, to optimize performance of their programs, but rarely, if ever, are they concerned with engineering aspects.

The goal of this chapter is to introduce the basic concepts of computer architecture and organization in order to allow computational scientists to recognize when programs are not as

be "send the instruction at location $M[1000]$ " or a disk controller's request might be "store the following block of data in locations $M[0]$ through $M[255]$."

Input/output (I/O) devices transfer information without altering it between the external world and one or more internal components. I/O devices can be secondary memories, for example disks and tapes, or devices used to communicate directly with users, such as video displays, keyboards, and mice.

The communication channels that tie the system together can either be simple links that connect two devices or more complex *switches* that interconnect several components and allow any two of them to communicate at a given point in time. When a switch is configured to allow two devices to exchange information, all other devices that rely on the switch are *blocked*, i.e. they must wait until the switch can be reconfigured.

A common convention used in drawing simple "stick figures" of computer systems is the PMS notation [32]. In a PMS diagram each major component is represented by a single letter, e.g. P for processor, M for memory, or S for switch. A subscript on a letter distinguished different types of components, e.g. M_p for primary memory and M_c for cache memory. Lines connecting two components represent links, and lines connecting more than two components represent a switch. Although they are primitive and might appear at first glance to be too simple, PMS diagrams convey quite a bit of information and have several advantages, not the least of which is they are independent of any particular manufacturer's notations.

As an example of a PMS diagram and a relatively simple computer architecture, Figure 1 shows the major components of the original Apple Macintosh personal computer. The first thing one notices is a single communication channel, known as the *bus*, that connects all the other major components. Since the bus is a switch, only two of these components can communicate at any time. When the switch is configured for an I/O transfer, for example from main memory (M_p) to the disk (via K_{disk}), the processor is unable to fetch data or instructions and remains idle. This organization is typical of personal computers and low end workstations; mainframes, supercomputers, and other high performance systems have much richer (and thus more expensive) structures for connecting I/O devices to internal main memory that allow the processor to keep working at full speed during I/O operations.

2.1 Processors

The operation of a processor is characterized by a *fetch-decode-execute* cycle. In the first phase of the cycle, the processor fetches an instruction from memory. The address of the instruction to fetch is stored in an internal register² named the *program counter*, or PC. As the processor is waiting for the memory to respond with the instruction, it increments the PC. This means the fetch phase of the next cycle will fetch the instruction in the next sequential location in memory (unless the PC is modified by a later phase of the cycle).

In the decode phase the processor stores the information returned by the memory in another internal register, known as the instruction register, or IR. The IR now holds a single

²A register is a small piece of memory large enough to hold a single number

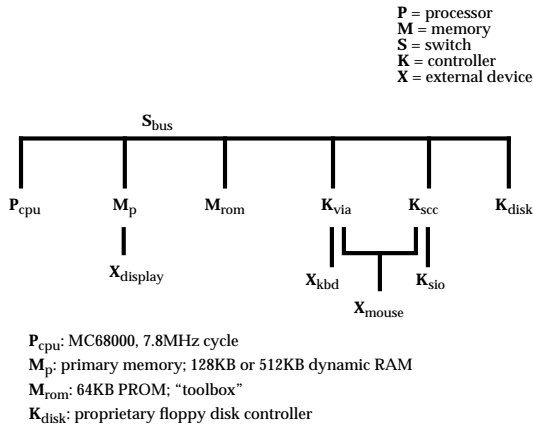


Figure 1: PMS Diagram of the Apple Macintosh

machine instruction, encoded as a binary number. The processor decodes the value in the IR in order to figure out which operations to perform in the next stage.

In the execution stage the processor actually carries out the instruction. This step often requires further memory operations; for example, the instruction may direct the processor to fetch two operands from memory, add them, and store the result in a third location (the addresses of the operands and the result are also encoded as part of the instruction). At the end of this phase the machine starts the cycle over again by entering the fetch phase for the next instruction.

Instructions can be classified as one of three major types: arithmetic/logic, data transfer, and control. Arithmetic and logic instructions apply primitive functions of one or two arguments, for example addition, multiplication, or logical AND. In some machines the arguments are fetched from main memory and the result is returned to main memory, but more often the operands are all in registers inside the CPU. Most machines have a set of general purpose registers that can be used for holding such operands. For example the HP-PA processor in Hewlett-Packard workstations has 32 such registers, each of which holds a single number.

The data transfer instructions move data from one location to another, for example between registers, or from main memory to a register, or between two different memory locations. Data transfer instructions are also used to initiate I/O operations.

Control instructions modify the order in which instructions are executed. They are used to construct loops, if-then-else constructs, etc. For example, consider the following DO loop in Fortran:

```
DO 10 I=1,5
...
10 CONTINUE
```

To implement the bottom of the loop (at the CONTINUE statement) there might be an arithmetic instruction that adds 1 to I, followed by a control instruction that compares I to 5 and branches to the top of the loop if I is less than or equal to 5. The branch operation is performed by simply setting the PC to the address of the instruction at the top of the loop.

The timing of the fetch, decode, and execute phases depends on the internal construction of the processor and the complexity of the instructions it executes. The quantum time unit for measuring operations is known as a *clock cycle*. The logic that directs operations within a processor is controlled by an external clock, which is simply a circuit that generates a square wave with a fixed period. The number of clock cycles required to carry out an operation determines the amount of time it will take.

One cannot simply assume that if a multiplication can be done in t_m nanoseconds then it will take $n \cdot t_m$ nanoseconds to perform n multiplications or that if a branch instruction takes t_b nanoseconds the next instruction will begin execution t_b nanoseconds following the branch. The actual timings depend on the organization of the memory system and the communication channels that connect the processor to the memory; these are the topics of the next two sections.

2.2 Memories

Memories are characterized by their function, capacity, and response times. Operations on memories are called *reads* and *writes*, defined from the perspective of a processor or other device that uses a memory: a read transfers information from the memory to the other device, and a write transfers information into the memory. A memory that performs both reads and writes is often just called a RAM, for *random access memory*. The term "random access" means that if location $M[x]$ is accessed at time t , there are no restrictions on the address of the item accessed at time $t + 1$. Other types of memories commonly used in systems are read-only memory, or ROM, and programmable read-only memory, or PROM (information in a ROM is set when the chips are designed; information in a PROM can be written later, one time only, usually just before the chips are inserted into the system). For example, the Apple Macintosh, shown in Figure 1, had a PROM called the "toolbox" that contained code for commonly used operating system functions.

The smallest unit of information is a single bit, which can have one of two values. The capacity of an individual memory chip is often given in terms of bits. For example one might have a memory built from 64Kb (64 kilobit) chips. When discussing the capacity of an entire memory system, however, the preferred unit is a *byte*, which is commonly accepted to be 8 bits of information. Memory sizes in modern systems range from 4MB (megabytes) in small personal computers up to several billion bytes (gigabytes, or GB) in large high-performance systems. Note the convention that lower case b is the abbreviation for bit and upper case B is the symbol for bytes.

The performance of a memory system is defined by two different measures, the *access time* and the *cycle time*. Access time, also known as *response time* or *latency*, refers to how quickly the memory can respond to a read or write request. Several factors contribute to the access time of a memory system. The main factor is the physical organization of the memory chips used in the system. This time varies from about 80 ns in the chips used in personal computers to 10 ns or less for chips used in caches and buffers (small, fast memories used for temporary storage, described in more detail below). Other factors are harder to measure. They include the overhead involved in selecting the right chips (a complete memory system will have hundreds of individual chips), the time required to forward a request from the processor over the bus to the memory system, and the time spent waiting for the bus to finish a previous transaction before initiating the processor's request. The bottom line is that the response time for a memory system is usually much longer than the access time of the individual chips.

Memory cycle time refers to the minimum period between two successive requests. For various reasons the time separating two successive requests is not always 0, i.e. a memory with a response time of 80 ns cannot satisfy a request every 80 ns. A simple, if old, example of a memory with a long cycle time relative to its access time is the magnetic core used in early mainframe computers. In order to read the value stored in memory, an electronic pulse was sent along a wire that was threaded through the core. If the core was in a given state, the pulse induced a signal on a second wire. Unfortunately the pulse also erased the information that used to be in memory, i.e. the memory had a destructive read-out. To get

around this problem designers built memory systems so that each time something was read a copy was immediately written back. During this write the memory cell was unavailable for further requests, and thus the memory had a cycle time that was roughly twice as long as its access time. Some modern semiconductor memories have destructive reads, and there may be several other reasons why the cycle time for a memory is longer than the access time.

Although processors have the freedom to access items in a RAM in any order, in practice the pattern of references is not random, but in fact exhibits a structure that can be exploited to improve performance. The fact that instructions are stored sequentially in memory (recall that unless there is a branch, PC is incremented by one each time through the fetch-decode-execute cycle) is one source of regularity. What this means is that if a processor requests an instruction from location x at time t , there is a high probability that it will request an instruction from location $x + 1$ in the near future at time $t + \delta$. References to data also show a similar pattern; for example if a program updates every element in a vector inside a small loop the data references will be to $v[0]$, $v[1]$, \dots . This observation that memory references tend to cluster in small groups is known as *locality of reference*.

Locality of reference can be exploited in the following way. Instead of building the entire memory out of the same material, construct a hierarchy of memories, each with different capacities and access times. At the top of the hierarchy there will be a small memory, perhaps only a few KB, built from the fastest chips. The bottom of the hierarchy will be the largest but slowest memory. The processor will be connected to the top of the hierarchy, i.e. when it fetches an instruction it will send its request to the small, fast memory. If this memory contains the requested item, it will respond, and the request is satisfied. If a memory does not have an item, it forwards the request to the next lower level in the hierarchy.

The key idea is that when the lower levels of the hierarchy send a value from location x to the next level up, they also send the contents of $x + 1$, $x + 2$ etc. If locality of reference holds, there is a high probability there will soon be a request for one of these other items; if there is, that request will be satisfied immediately by the upper level memory.

The following terminology is used when discussing hierarchical memories:

- The memory closest to the processor is known as a *cache*. Some systems have separate caches for instructions and data, in which case it has a *split cache*. An *instruction buffer* is a special cache for instructions that also performs other functions that make fetching instructions more efficient.
- The main memory is known as the *primary memory*.
- The low end of the hierarchy is the *secondary memory*. It is often implemented by a disk, which may or may not be dedicated to this purpose.
- The unit of information transferred between items in the hierarchy is a *block*. Blocks transferred to and from cache are also known as *cache lines*, and units transferred between primary and secondary memory are also known as *pages*.

- Eventually the top of the hierarchy will fill up with blocks transferred from the lower levels. A *replacement strategy* determines which block currently in a higher level will be removed to make room for the new block. Common replacement strategies are random replacement (throw out any current block at random), first-in-first-out (FIFO; replace the block that has been in memory the longest), and least recently used (LRU; replace the block that was last referenced the furthest in the past).
- A request that is satisfied is known as a *hit*, and a request that must be passed to a lower level of the hierarchy is a *miss*. The percentage of requests that result in hits determines the *hit rate*. The hit rate depends on the size and organization of the memory and to some extent on the replacement policy. It is not uncommon to have a hit rate near 99% for caches on workstations and mainframes.

The performance of a hierarchical memory is defined by the *effective access time*, which is a function of the hit ratio and the relative access times between successive levels of the hierarchy. For example, suppose the cache access time is 10ns, main memory access time is 100ns, and the cache hit rate is 98%. Then the average time for the processor to access an item in memory is

$$\begin{aligned} t_{eff} &= 0.98 \cdot t_{cache} + 0.02 \cdot t_{main} \\ &= 11.8ns \end{aligned}$$

Over a long period of time the system performs as if it had a single large memory with an 11.8ns cycle time, thus the term “effective access time.” With a 98% hit rate the system performs nearly as well as if the entire memory was constructed from the fast chips used to implement the cache, i.e. the average access time is 11.8ns, even though most of the memory is built using less expensive technology that has an access time of 100ns.

Although a memory hierarchy adds to the complexity of a memory system, it does not necessarily add to the latency for any particular request. There are efficient hardware algorithms for the logic that looks up addresses to see if items are present in a memory and to help implement replacement policies, and in most cases these circuits can work in parallel with other circuits so the total time spent in the fetch-decode-execute cycle is not lengthened.

2.3 Buses

A bus is used to transfer information between several different modules. Small and mid-range computer systems, such as the Macintosh shown in Figure 1 have a single bus connecting all major components. Supercomputers and other high performance machines have more complex interconnections, but many components will have internal buses.

Communication on a bus is broken into discrete *transactions*. Each transaction has a sender and receiver. In order to initiate a transaction, a module has to gain control of the bus and become (temporarily, at least) the bus *master*. Often several devices have the

ability to become the master; for example, the processor controls transactions that transfer instructions and data between memory and CPU, but a disk controller becomes the bus master to transfer blocks between disk and memory. When two or more devices want to transfer information at the same time, an *arbitration protocol* is used to decide which will be given control first. A protocol is a set of signals exchanged between devices in order to perform some task, in this case to agree which device will become the bus master.

Once a device has control of the bus, it uses a *communication protocol* to transfer the information. In an *asynchronous* (unclocked) protocol the transfer can begin at any time, but there is some overhead involved in notifying potential receivers that information needs to be transferred. In a *synchronous* protocol transfers are controlled by a global clock and begin only at well-known times.

The performance of a bus is defined by two parameters, the transfer time and the overall *bandwidth* (sometimes called *throughput*). Transfer time is similar to latency in memories: it is the amount of time it takes for data to be delivered in a single transaction. For example, the transfer time defines how long a processor will have to wait when it fetches an instruction from memory. Bandwidth, expressed in units of bits per second (bps), measures the capacity of the bus. It is defined to be the product of the number of bits that can be transferred in parallel in any one transaction by the number of transactions that can occur in one second. For example, if the bus has 32 data lines and can deliver 1,000,000 packets per second, it has a bandwidth of 32Mbps.

At first it may seem these two parameters measure the same thing, but there are subtle differences. The transfer time measures the delay until a piece of data arrives. As soon as the data is present it may be used while other signals are passed to complete the communication protocol. Completing the protocol will delay the next transaction, and bandwidth takes this extra delay into account. Another factor that distinguishes the two is that in many high performance systems a block of information can be transferred in one transaction; in other words, the communication protocol may say “send n items from location x .” There will be some initial overhead in setting up the transaction, so there will be a delay in receiving the first piece of data, but after that information will arrive more quickly.

Bandwidth is a very important parameter. It is also used to describe processor performance, when we count the number of instructions that can be executed per unit time, and the performance of networks.

2.4 I/O

Many computational science applications generate huge amounts of data which must be transferred between main memory and I/O devices such as disk and tape. We will not attempt to characterize file I/O in this chapter since the devices and their connections to the rest of the system tend to be idiosyncratic. If your application needs to read or write large data files you will need to learn how your system organizes and transfers files and tune your application to fit that system. It is worth reiterating, though, that performance is measured in terms of bandwidth: what counts is the volume of data per unit of time that can be moved

into and out of main memory.

The rest of this section contains a brief discussion of video displays. These output devices and their capabilities also vary from system to system, but since scientific visualization is such a prominent part of this book we should introduce some concepts and terminology for readers who are not familiar with video displays.

Most users who generate high quality images will do so on workstations configured with extra hardware for creating and manipulating images. Almost every workstation manufacturer includes in its product line versions of their basic systems that are augmented with extra processors that are dedicated to drawing images. These extra processors work in parallel with the main processor in the workstation. In most cases data generated on a supercomputer is saved in a file and later viewed on a video console attached to a graphics workstation. However there are situations that make use of high bandwidth connections from supercomputers directly to video displays; these are useful when the computer is generating complex data that should be viewed in “real time.” For example, a demonstration program from Thinking Machines, Inc. allows a user to move a mouse over the image of a fluid moving through a pipe. When the user pushes the mouse button, the position of the mouse is sent to a parallel processor which simulates the path of particles in a turbulent flow at this position. The results of the calculations are sent directly to the video display, which shows the new positions of the particles in real time. The net effect is as if the user is holding a container of fluid that is being poured into the pipe.

There are many different techniques for drawing images with a computer, but the dominant technology is based on a *raster scan*. A beam of electrons is directed at a screen that contains a quick-fading phosphor. The beam can be turned on and off very quickly, and it can be bent in two dimensions via magnetic fields. The beam is swept from left to right (from the user’s point of view) across the screen. When the beam is on, a small white dot will appear on the screen where the beam is aimed, but when it is off the screen will remain dark. To paint an image on the entire screen, the beam is swept across the top row; when it reaches the right edge, it is turned off, moved back to the left and down one row, and then swept across to the right again. When it reaches the lower right corner, the process repeats again in the upper left corner.

The number of times per second the full screen is painted determines the *refresh rate*. If the rate is too low, the image will flicker, since the bright spots on the phosphor will fade before the gun comes back to that spot on the next pass. Refresh rates vary from 30 times per second up to 60 times per second.

The individual locations on a screen that can be either painted or not are known as *pixels* (from “picture cell”). The *resolution* of the image is the number of pixels per inch. A high resolution display will have enough pixels in a given area that from a reasonable distance (an arm’s length away) the gaps between pixels are not visible and a sequence of pixels that are all on will appear to be a continuous line. A common screen size is 1280 pixels across and 1024 pixels high on a 16” or 19” monitor.

The controller for the electron gun decides whether a pixel will be black or white by reading information from a memory that has one bit per pixel. If the bit is a 1, the pixel

will be painted, otherwise it will remain dark. From the PMS diagram in Figure 1 you can see that the display memory on the Macintosh was part of the main memory. The operating system set aside a portion of the main memory for displays, and all an application had to do to paint something on the screen was to write a bit pattern into this portion of memory. This was an economical choice for the time (early 1980s), but it came at the cost of performance: the processor and video console had to alternate accesses to memory. During periods when the electron gun was being moved back to the upper left hand corner, the display did not access memory, and the processor was able to run at full speed. Once the gun was positioned and ready for the next scan line, however, the processor and display went back to alternating memory cycles.

With the fall in memory prices and the rising demand for higher performance, modern systems use a dedicated memory known as a *frame buffer* for holding bit patterns that control the displays. On inexpensive systems the main processor will compute the patterns and transfer them to the frame buffer. On high performance systems, though, the main processor sends information to the “graphics engine”, a dedicated processor that performs the computations. For example, if the user wants to draw a rectangle, the CPU can send the coordinates to the graphics processor, and the latter will figure out which pixels lie within the rectangle and turn on the corresponding bits in the frame buffer. Sophisticated graphics processors do all the work required in complex shading, texturing, overlapping of objects (deciding what is visible and what is not), and other operations required in 3D images.

The discussion so far has dealt only with black and white images. Color displays are based on the same principles: a raster scan illuminates regions on a phosphor, with the information that controls the display coming from a frame buffer. However, instead of one gun there are three, one for each primary color. When combining light, the primary colors are red, green, and blue, which is why these displays are known as RGB monitors.³ Since we need to specify whether or not each gun should be on for each pixel, the frame buffer will have at least three bits per pixel. To have a wide variety of colors, though, it is not enough just to turn a gun on or off; we need to control its intensity. For example, a violet color can be formed by painting a pixel with the red gun at 61% of full intensity, green at 24%, and blue at 80%.

Typically a system will divide the range of intensities into 256 discrete values, which means the intensity can be represented by an 8-bit number. 8 bits times 3 guns means 24 bits are required for each pixel. Recall that high resolution displays have 1024 rows of 1280 pixels each, for a total of 1.3 million pixels. Dedicating 24 bits to each pixel would require almost 32MB of RAM for the frame buffer alone. What is done instead is to create a *color map* with a fixed number of entries, typically 256. Each entry in the color map is a full 24

³Combining primaries to make secondary and other colors is very different for additive, light-based colors than it is for subtractive, paint-based colors. With paints, the primaries are red, blue, and yellow. As an example, combining red and yellow paint in equal proportions creates an orange paint. To create an orange light, however, one needs to combine two parts red, one part green, and no blue. X windows users who are curious to see how RGB primaries are combined to make their favorite colors can look at the system color database, usually in a file named `/usr/lib/X11/rgb.txt`.

bits wide. Each pixel only needs to identify a location in the map that contains its color, and since a color map of 256 entries requires only 8 bits per pixel to specify one of the entries there is a savings of 16 bits per pixel. The drawback is that only 256 different colors can be displayed in any one image, but this is enough for all applications except those that need to create highly realistic images.

2.5 Operating Systems

The user's view of a computer system is of a complex set of services that are provided by a combination of hardware (the architecture and its organization) and software (the operating system). Attributes of the operating system also affect the performance of user programs.

Operating systems for all but the simplest personal computers are *multi-tasking* operating systems. This means the computer will be running several jobs at once. A *program* is a static description of an algorithm. To run a program, the system will decide how much memory it needs and then start a *process* for this program; a process (also known as a *task*) can be viewed as a dynamic copy of a program. For example, the C compiler is a program. Several different users can be compiling their code at the same time; there will be a separate process in the system for each of these invocations of the compiler.

Processes in a multi-tasking operating system will be in one of three states. A process is *active* if the CPU is executing the corresponding program. In a single processor system there will be only one active process at any time. A process is idle if it is waiting to run. In order to allocate time on the CPU fairly to all processes, the operating system will let a process run for a short time (known as a *time slice*; typically around 20ms) and then interrupt it, change its status to idle, and install one of the other idle tasks as the new active process. The previous task goes to the end of a *process queue* to wait for another time slice.

The third state for a process is *blocked*. A blocked process is one that is waiting for some external event. For example, if a process needs a piece of data from a file, it will call the operating system routine that retrieves the information and then voluntarily give up the remainder of its time slice. When the data is ready, the system changes the process' state from blocked to idle, and it will be resumed again when its turn comes.

The predominant operating systems for workstations is Unix, developed in the 1970s at Bell Labs and made popular in the 1980s by the University of California at Berkeley. Even though there may be just one user, and that user is executing only one program (e.g. a text editor), there will be dozens of tasks running. Many Unix services are provided by small systems programs known as daemons that are dedicated to one special purpose. There are *daemons* for sending and receiving mail, using the network to find files on other systems, and several other jobs.

The fact that there may be several processes running in a system at the same time as your computational science application has ramifications for performance. One is that it makes it slightly more difficult to measure performance. You cannot simply start a program, look at your watch, and then look again when the program stops to measure the time spent. This measure is known as *real time* or "wall-clock time," and it depends as much on the

number of other processes in the system as it does on the performance of your program. Your program will take longer to run on a heavily-loaded system since it will be competing for CPU cycles with those other jobs. To get an accurate assessment of how much time is required to run your program you need to measure CPU *time*. Unix and other operating systems have system routines that can be called from an application to find out how much CPU time has been allocated to the process since it was started.

Another impact of having several other jobs in the process queue is that as they are executed they work themselves into the cache, displacing your program and data. During your application's time slice its code and data will fill up the cache. But when the time slice is over and a daemon or other user's program runs, its code and data will soon replace yours, so that when yours resumes it will have a higher miss rate until it reloads the code and data it was working on when it was interrupted. This period during which your information is being moved back into the cache is known as a *reload transient*. The longer the interval between time slices and the more processes that run during this interval the longer the reload transient.

Supercomputers and parallel processors also use variants of Unix for their runtime environments. You will have to investigate whether or not daemons run on the main processor or a "front end" processor and how the operating system allocates resources. As an example of the range of alternatives, on an Intel Paragon XPS with 56 processors some processors will be dedicated to system tasks (e.g. file transfers) and the remainder will be split among users so that applications do not have to share any one processor. The MasPar 1104 consists of a front-end (a DEC workstation) that handles the system tasks and 4096 processors for user applications. Each processor has its own 64KB RAM. More than one user process can run at any one time, but instead of allocating a different set of processors to each job the operating system divides up the memory. The memory is split into equal size partitions, for example 8KB, and when a job starts the system figures out how many partitions it needs. All 4096 processors execute that job, and when the time slice is over they all start working on another job in a different set of partitions.

2.6 Data Representations

Another important interaction between user programs and computer architecture is in the representation of numbers. This interaction does not affect performance as much as it does portability. Users must be extremely careful when moving programs and/or data files from one system to another because numbers and other data are not always represented the same way. Recently programming languages have begun to allow users to have more control over how numbers are represented and to write code that does not depend so heavily on data representations that it fails when executed on the "wrong" system.

The binary number system is the starting point for representing information. All items in a computer's memory - numbers, characters, instructions, etc. - are represented by strings of 1's and 0's. These two values designate one of two possible states for the underlying physical memory. It does not matter to us which state corresponds to 1 and which corresponds to 0,

or even what medium is used. In an electronic memory, 1 could stand for a positively charged region of semiconductor and 0 for a neutral region, or on a device that can be magnetized a 1 would represent a portion of the surface that has a flux in one direction, while a 0 would indicate a flux in the opposite direction. It is only important that the mapping from the set {1,0} to the two states be consistent and that the states can be detected and modified at will.

Systems usually deal with fixed-length strings of binary digits. The smallest unit of memory is a single *bit*, which holds a single binary digit. The next largest unit is a *byte*, now universally recognized to be eight bits (early systems used anywhere from six to eight bits per byte). A *word* is 32 bits long in most workstations and personal computers, and 64 bits in supercomputers. A *double word* is twice as long as a single word, and operations that use double words are said to be *double precision* operations.

Storing a positive integer in a system is trivial: simply write the integer in binary and use the resulting string as the pattern to store in memory. Since numbers are usually stored one per word, the number is padded with leading 0's first. For example, the number 52 is represented in a 16-bit word by the pattern 0000000000110100.

The meaning of an n -bit string s when it is interpreted as a binary number is defined by the formula, $x = s_1 \times 2^{n-1} + s_2 \times 2^{n-2} + \dots + s_{n-1} \times 2^1 + s_n \times 2^0$, i.e. bit number i has weight:

$$x = \sum_{i=1}^n s_i \times 2^{n-i}$$

Compiler writers and assembly language programmers often take advantage of the binary number system when implementing arithmetic operations. For example, if the pattern of bits is "shifted left" by one, the corresponding number is multiplied by two. A left shift is performed by moving every bit left and inserting 0's on the right side. In an 8-bit system, for example, the pattern 00000110 represents the number 6; if this pattern is shifted left, the resulting pattern is 00001100, which is the representation of the number 12. In general, shifting left by n bits is equivalent to multiplying by 2^n .

Shifts such as these can be done in one machine cycle, so they are much faster than multiplication instructions, which usually takes several cycles. Other "tricks" are using a right shift to implement integer division by a power of 2, in which the result is an integer and the remainder is ignored (e.g. $15 \div 4 = 3$) and taking the modulus or remainder with respect to a power of 2 (see problem 8).

A fundamental relationship about binary patterns is that there are 2^n distinct n -digit strings. For example, for $n = 8$ there are $2^8 = 256$ different strings of 1's and 0's. From this relationship it is easy to see that the largest integer that can be stored in an n -bit word is $2^n - 1$: the 2^n patterns are used to represent the 2^n integers in the interval $[0, \dots, 2^n - 1]$.

An *overflow* occurs when a system generates a value greater than the largest integer. For example, in a 32-bit system, the largest positive integer is $2^{32} - 1 = 4,294,967,295$. If a program tries to add 3,000,000,000 and 2,000,000,000 it will cause an overflow. Right away we can see one source of problems that can arise when moving a program from one system

to another: if the word size is smaller on the new system a program that runs successfully on the original system may crash with an overflow error on the new system.

There are two different techniques for representing negative values. One method is to divide the word into two *fields*, i.e. represent two different types of information within the word. We can use one field to represent the sign of the number, and the other field to represent the value of the number. Since a number can be just positive or negative, we need only one bit for the sign field. Typically the leftmost bit represents the sign, with the convention that a 1 means the number is negative and a 0 means it is positive. This type of representation is known as a *sign-magnitude* representation, after the names of the two fields. For example, in a 16-bit sign-magnitude system, the pattern 10000000111111 represents the number and the pattern 00000000000001 represents +5.

The other technique for representing both positive and negative integers is known as *two's complement*. It has two compelling advantages over the sign-magnitude representation, and is now universally used for integers, but as we will see below sign-magnitude is still used to represent real numbers. The two's complement method is based on the fact that binary arithmetic in fixed-length words is actually arithmetic over a finite cyclic group. If we ignore overflows for a moment, observe what happens when we add 1 to the largest possible number in an n -bit system (this number is represented by a string of n 1's):

$$\begin{array}{r} 1111 \dots 1111 \\ + \\ 1 \\ \hline 10000 \dots 0000 \end{array}$$

The result is a pattern with a leading 1 and n 0's. In an n -bit system only the low order n bits of each result are saved, so this sum is functionally equivalent to 0. Operations that lead to sums with very large values "wrap around" to 0, i.e. the system is a finite cyclic group. Operations in this group are defined by arithmetic modulo 2^n .

For our purposes, what is interesting about this type of arithmetic is that 2^n , which is represented by a 1 followed by n 0's, is equivalent to 0, which means $2^n - x = -x$ for all x between 0 and $2^n - 1$. A simple "trick" that has its roots in this fact can be applied to the bit pattern of a number in order to calculate its additive inverse: if we invert every bit (turn a 1 into a 0 and vice versa) in the representation of a number x and then add 1, we come up with the representation of $-x$. For example, the representation of 5 in an 8-bit system is 00000101. Inverting every bit and adding 1 to the result gives the pattern 11111011. This is also the representation of 251, but in arithmetic modulo 2^8 we have so this pattern is a perfectly acceptable representation of -5 (see problem 7).

In practice we divide all n -bit patterns into two groups. Patterns that begin with 0 represent the positive integers $0 \leq x \leq 2^{n-1} - 1$ and patterns beginning with 1 represent the negative integers $-2^{n-1} \leq x < 0$. To determine which integer is represented by a pattern that begins with a 1, compute its complement (invert every bit and add 1). For example, in an 8-bit two's complement system the pattern 11100001 represents -5, since the complement is $00011110 + 1 = 00011111_2 = 31_{10}$. Note that the leading bit determines the sign, just as in

a sign-magnitude system, but one cannot simply look at the remaining bits to ascertain the magnitude of the number. In a sign-magnitude system, the same pattern represents -97 .

The first step in defining a representation for real numbers is to realize that binary notation can be extended to cover negative powers of two, e.g. the string “110.101” is interpreted as

$$1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 6.625$$

Thus a straightforward method for representing real numbers would be to specify some location within a word as the “binary point” and give bits to the left of this location weights that are positive powers of two and bits to the right weights that are negative powers of two. For example, in a 16-bit word, we can dedicate the rightmost 5 bits for the fraction part and the leftmost 11 bits for the whole part. In this system, the representation of 6.625 is 0000000011010100 (note there are leading 0’s to pad the whole part and trailing 0’s to pad the fraction part). This representation, where there is an implied binary point at a fixed location within the word, is known as a *fixed point* representation.

There is an obvious tradeoff between *range* and *precision* in fixed point representations. n bits for the fraction part means there will be 2^n numbers in the system between any two successive integers. With 5 bit fractions there are 32 numbers in the system between any two integers; e.g. the numbers between 5 and 6 are $5\frac{1}{32}$ (5.03125), $5\frac{2}{32}$ (5.03125), etc. To allow more precision, i.e. smaller divisions between successive numbers, we need more bits in the fraction part. The number of bits in the whole part determines the magnitude of the largest positive number we can represent, just as it does for integers. With 11 digits in the whole part, as in the example above, the largest number we can represent in 16 bits is $111111111.11111_2 = 2047.96875_{10}$. Moving one bit from the whole part to the fraction part in order to increase precision cuts the range in half, and the largest number is now $111111111.11111_2 = 1023.984375_{10}$.

To allow for a larger range without sacrificing precision, computer systems use a technique known as *floating point*. This representation is based on the familiar “scientific notation” for expressing both very large and very small numbers in a concise format as the product of a small real number and a power of 10, e.g. 6.022×10^{23} . This notation has three components: a *base* (10 in this example); an *exponent* (in this case 23); and a *mantissa* (6.022). In computer systems, the base is either 2 or 16. Since it never changes for any given computer system it does not have to be part of the representation, and we need only two fields to specify a value, one for the mantissa and one for the exponent.

As an example of how a number is represented in floating point, consider again the number 6.625. In binary, it is

$$110.101 \times 2^0 = 1.10101 \times 2^2$$

If a 16-bit system has a 10-bit mantissa and 6-bit exponent, the number would be represented by the string 1101010000 000010. The mantissa is stored in the first ten bits (padded on the right with trailing 0’s), and the exponent is stored in the last six bits.



Figure 2: Distribution of Floating Point Numbers

As the above example illustrates, computers transform the numbers so the mantissa is a manageable number. Just as 6.022×10^{23} is preferred to 60.22×10^{22} or 0.6022×10^{24} in scientific notation, in binary the mantissa should be between 1.000... and 1.111... . When the mantissa is in this range it is said to be *normalized*. The definition of the normal form varies from system to system, e.g. in some systems a normalized mantissa is between 0.1000... and 0.1111... .

Since we need to represent both positive and negative real numbers, the complete representation for a real number in a floating point format has three fields: a one-bit sign, a fixed number of bits for the mantissa, and the remainder of the bits for the exponent. Note that the exponent is an integer, and that this integer can be either positive or negative, e.g. we will want to represent very small numbers such as 4.1×10^{-15} . Any method such as two’s complement that can represent both positive and negative integers can be used within the exponent field. The sign bit at the front of the number determines the sign of the entire number, which is independent of the sign of the exponent, e.g. it indicates whether the number is 4.1×10^{-15} or -4.1×10^{-15} .

In the past every computer manufacturer used their own floating point representation, which made it a nightmare to move programs and datasets from one system to another. A recent IEEE standard is now being widely adopted and will add stability to this area of computer architecture. For 32-bit systems, the standard calls for a 1-bit sign, 8-bit exponent, and 23-bit mantissa. The largest number that can be represented is $2^{128} \approx 10^{38}$, and the smallest positive number (closest to 0.0) is $2^{-150} \approx 10^{-47}$. Details of the standard are presented in an appendix to this chapter.

Figure 2 illustrates the numbers that can be stored in a typical computer system with a floating point representation. The figure shows three disjoint regions: positive numbers $\varepsilon \leq \omega$, 0.0, and negative numbers $-\omega \leq n \leq -\varepsilon$. ω is the largest number that can be stored in the system; in the IEEE standard representation $\omega = 10^{38}$. ε is the smallest positive number, which is 10^{-47} in the IEEE standard.

Programmers need to be aware of several important attributes of the floating point representation that are illustrated by this figure. The first is the magnitude of the range between $-\omega$ and ω . There are about 10^{38} integers in this range. However there are only $2^{32} = 10^9$ different 32-bit patterns. What this means is there are numbers in the range that do not have representations. Whenever a calculation results in one of these numbers, a round-off error will occur when the system approximates the result by the nearest (we hope) representable

number. The arithmetic circuitry will produce a binary pattern that is close to the desired result, but not an exact representation. An interesting illustration of just how common these round-off errors are is the fact that 1 does not have a finite representation in binary, but is instead the infinitely repeating pattern 0.0001100110011...².

The next important point is that there is a gap between ε , the smallest positive number, and 0.0. A round-off error in a calculation that should produce a small non-zero value but instead results in 0.0 is called an *underflow*. One of the strengths of the IEEE standard is that it allows a special *denormalized* form for very small numbers in order to stave off underflows as long as possible. This is why the exponent in the largest and smallest positive numbers are not symmetrical. Without denormalized numbers, the smallest positive number in the IEEE standard would be around 10^{-38} .

Finally, and perhaps most important, is the fact that the numbers that can be represented are not distributed evenly throughout the range. Representable numbers are very dense close to 0.0, but then grow steadily further apart as they increase in magnitude. The dark regions in Figure 2 correspond to parts of the number line where representable numbers are packed close together. It is easy to see why the distribution is not even by asking what two numbers are represented by two successive values of the mantissa for any given exponent. To make the calculations easier, suppose we have a 16-bit system with a 7-bit mantissa and 8-bit exponent. No matter what the exponent is, the distance between any two successive values of the mantissa, e.g. between 0.1110000_2 and 0.1110001_2 , will be $0.0000001_2 \approx 0.0078_{10}$. For numbers closest to 0.0, the exponent will be a negative number, e.g. -100 , and the distance between two successive floating point numbers will be $0.0000001_2 \times 2^{100} \approx 0.0078 \times 10^{-30} = 7.8 \times 10^{-33}$. At the other end of the scale, when exponents are large, the distance between two numbers will be approximately 2^{100} , namely $0.0000001_2 \times 2^{100} \approx 0.0078 \times 10^{60} = 7.8 \times 10^{47}$.

2.7 Performance Models

The most widely recognized aspect of a machine’s internal organization that relates to performance is the clock cycle time, which controls the rate of internal operations in the CPU (Section 2.1). A shorter clock cycle time, or equivalently a larger number of cycles per second, implies more operations can be performed per unit time.

For a given architecture, it is often possible to rank systems according to their clock rates. For example, the HP 9000/725 and 9000/735 workstations have basically the same architecture, meaning they have the same instruction set and, in general, appear to be the same system as far as compiler writers are concerned. The 725 has a 66MHz clock, while the 735 has a 99MHz clock, and indeed the 735 has a higher performance on most programs.

There are several reasons why simply comparing clock cycle times is an inadequate measure of performance. One reason is that processors don’t operate “in a vacuum”, but rely on memories and buses to supply information. The size and access times of the memories and the bandwidth of the bus all play a major role in performance. It is very easy to imagine a program that requires a large amount of memory running faster on an HP 725 that has

Machine A			Machine B		
L:	load X,V[i]	(2)	L:	load X,V[i]	(2)
	mpy 3,X	(4)		mov X,Y	(2)
	store X,V[i]	(2)		shl X,1	(2)
	dbr L,i	(2)		add Y,X	(2)
				store X,V[i]	(2)
				dbr L,i	(2)
total cycles:		10	total cycles:		12

Table 1:

a larger cache and more main memory than a 735. We will return to the topic of memory organization and processor-memory interconnection in later sections on vector processors and parallel processors since these two aspects of systems organization are even more crucial for high performance in those systems.

A second reason clock rate by itself is an inadequate measure of performance is that it doesn’t take into account what happens during a clock cycle. This is especially true when comparing systems with different instruction sets. It is possible that a machine might have a lower clock rate, but because it requires fewer cycles to execute the same program it would have higher performance. For example, consider two machines, A and B, that are almost identical except that A has a multiply instruction and B does not. A simple loop that multiplies a vector by a scalar (the constant 3 in this example) is shown in the table below. The number of cycles for each instruction is given in parentheses next to the instruction.

The first instruction loads an element of the vector into an internal processor register X. Next, machine A multiplies the vector element by 3, leaving the result in the register. Machine B does the same operation by shifting and adding, i.e. $3x = 2x + x$. B copies the contents of X to another register Y, shifts X left one bit (which multiplies it by 2), and then adds Y, again leaving the result in X. Both machines then store the result back into the vector in memory and branch back to the top of the loop if the vector index is not at the end of the vector (the comparison and branch are done by the dbr instruction). Machine A might be slightly slower than B, but since it takes fewer cycles it will execute the loop faster. For example if A’s cycle time is 9 MHz (11μs per cycle) and B’s cycle time is 10 MHz (10μs per cycle) A will execute one pass through the loop in 1.1μs but B will require 1.2μs.

As a historical note, microprocessor and microcomputer designers in the 1970s tended to build systems with instruction sets like those of machine A above. The goal was to include instructions with a large “semantic content,” e.g., multiplication is relatively more complex than loading a value from memory or shifting a bit pattern. The payoff was in reducing the overhead to fetch instructions, since fewer instructions could accomplish the same job. By the 1980s, however, it became widely accepted that instruction sets such as those of machine B were in fact a better match for VLSI chip technology. The move toward simpler instructions became known as **RISC**, for Reduced Instruction Set Computer. A RISC has fewer instructions in its repertoire, but more importantly each instruction is very simple. The fact that operations are so simple and so uniform leads to some very powerful implementation techniques, such as pipelining, and opens up room on the processor chip for items such as on-chip caches or multiple functional units, e.g., a CPU that has two or more arithmetic units. We will discuss these types of systems in more detail later, in the section on superscalar designs (Section 3.5.2). Another benefit to simple instructions is that cycle times can also be much shorter; instead of being only moderately faster, e.g. 10MHz vs. 9MHz as in the example above, cycle times on RISC machines are often much faster, so even though they fetch and execute more instructions they typically outperform complex instruction set (CISC) machines designed at the same time.

In order to compare performance of two machines with different instruction sets, and even different styles of instruction sets (e.g. RISC vs. CISC), we can break the total execution time into constituent parts [11]. The total time to execute any given program is the product of the number of machine cycles required to execute the program and the processor cycle time:

$$T = n_c \times t_c$$

The number of cycles executed can be rewritten as the number of instructions executed times the average number of cycles per instruction:

$$T = n_i \times \frac{n_c}{n_i} \times t_c$$

The middle factor in this expression describes the average number of machine cycles the processor devotes to each instruction. It is the number of cycles per instruction, or **CPI**. The basic performance model for a single processor computer system is thus

$$T = n_i \times \text{CPI} \times t_c$$

where

$$\text{CPI} = \frac{n_c}{n_i}$$

The three factors each describe different attributes of the execution of a program. The number of instructions depends on the algorithm, the compiler, and to some extent the instruction set of the machine. Total execution time can be reduced by lowering the instruction

count, either through a better algorithm (one that executes an inner loop fewer times, for example), a better compiler (one that generates fewer instructions for the body of the loop), or perhaps by changing the instruction set so it requires fewer instructions to encode the same algorithm. As we saw earlier, however, a more compact encoding as a result of a richer instruction set does not always speed up a program since complex instructions require more cycles. The interaction between instruction complexity and the number of cycles to execute a program is very involved, and it is hard to predict ahead of time whether adding a new instruction will really improve performance.

The second factor in the performance model is CPI. At first it would seem this factor is simply a measure of the complexity of the instruction set: simple instructions require fewer cycles, so RISC machines should have lower CPI values. That view is misleading, however, since it concerns a static quantity. The performance equation describes the average number of cycles per instruction *measured during the execution of a program*. The difference is crucial. Implementation techniques such as pipelining allow a processor to overlap instructions by working on several instructions at one time. These techniques will lower CPI and improve performance since more instructions are executed in any given time period. For example, the average instruction in a system might require three machine cycles: one to fetch it from cache, one to fetch its operands from registers, and one to perform the operation and store the result in a register. Based on this static description one might conclude the CPI is 3.0, since each instruction requires three cycles. However, if the processor can juggle three instructions at once, for example by fetching instruction $i + 2$ while it is locating the operands for instruction $i + 1$ and executing instruction i , then the *effective* CPI observed during the execution of the program is just a little over 1.0 (Figure 3). Note that this is another illustration of the difference between speed and bandwidth. Overall performance of a system can be improved by increasing bandwidth, in this case by increasing the number of instructions that flow through the processor per unit time, without changing the execution time of the individual instructions.

The third factor in the performance model is the processor cycle time t_c . This is usually in the realm of computer engineering: a better layout of the components on the surface of the chip might shorten wire lengths and allow for a faster clock, or a different material (e.g. gallium arsenide vs. silicon based semiconductors) might have a faster switching time. However, the architecture can also affect cycle time. One of the reasons RISC is such a good fit for current VLSI technology is that if the instruction set is small, it requires less logic to implement. Less logic means less space on the chip, and smaller circuits run faster and consume less power [12]. Thus the design of the instruction set, the organization of pipelines, and other attributes of the architecture and its implementation can impact cycle time.

We conclude this section with a few remarks on some metrics that are commonly used to describe the performance of computer systems. **MIPS** stands for “millions of instructions per second.” With the variation in instruction styles, internal organization, and number of processors per system it is almost meaningless for comparing two systems. As a point of reference, the DEC VAX 11/780 executed approximately one million instructions per second. You may see a system described as having performance rated at “X VAX MIPS.” This is



A processor can overlap the execution of several instructions. In this example the first instruction is fetched during the first cycle. In the second cycle, instruction 1 is handed to the part of the processor that prepares operands and the second instruction is fetched from memory. In cycles 3 through n the processor is working on three instructions at a time. Note also that one instruction is completed every cycle. $n + 2$ cycles are required to execute n instructions, so the average number of cycles per instruction is $\text{CPI} = ((n + 2) / n) = 1.0$.

Figure 3: Pipelined execution

a measure of performance normalized to VAX 11/780 performance. What this means is someone ran a program on the VAX, then ran the same program on the other system, and the ratio is X . The term “native MIPS” refers to the number of millions of instructions of the machine’s own instruction set that can be executed per second.

MFLOPS (pronounced “megaflops”) stands for “millions of floating point operations per second.” This is often used as a “bottom-line” figure. If you know ahead of time how many operations a program needs to perform, you can divide the number of operations by the execution time to come up with a MFLOPS rating. For example, the standard algorithm for multiplying $n \times n$ matrices requires $2n^3 - n$ operations (n^2 inner products, with n multiplications and $n - 1$ additions in each product). Suppose you compute the product of two 100×100 matrices in 0.35 seconds. Your computer achieved

$$(2(100)^3 - 100) / 0.35 = 5,714,000 \text{ ops/sec} = 5.714 \text{ MFLOPS}$$

Obviously this type of comparison ignores the overhead involved in setting up loops, checking terminating conditions, and so on, but as a “bottom line” it gets to the point: what you care about (in this example) is how long it takes to multiply two matrices, and if that operation is a major component of your research it makes sense to compare machines by how fast they can multiply matrices. A standard set of reference programs known as LINPACK (linear algebra package) is often used to compare systems based on their MFLOPS ratings by measuring execution times for Gaussian elimination on 100×100 matrices [8].

The term “theoretical peak MFLOPS” refers to how many operations per second would be

possible if the machine did nothing but numerical operations. It is obtained by calculating the time it takes to perform one operation and then computing how many of them could be done in one second. For example, if it takes 8 cycles to do one floating point multiplication, the cycle time on the machine is 20 nanoseconds, and arithmetic operations are not overlapped with one another, it takes 160ns for one multiplication, and

$$\frac{1,000,000,000 \text{ nanosec}}{1 \text{ second}} \times \frac{1 \text{ multiplication}}{160 \text{ nanosec}} = 6.25 \times 10^6 \text{ multiplication/sec}$$

so the theoretical peak performance is 6.25 MFLOPS. Of course, programs are not just long sequences of multiply and add instructions, so a machine rarely comes close to this level of performance on any real program. Most machines will achieve less than 10% of their peak rating, but vector processors or other machines with internal pipelines that have an effective CPI near 1.0 can often achieve 70% or more of their theoretical peak on small programs.

Using metrics such as CPI, MIPS, or MFLOPS to compare machines depends heavily on the programs used to measure execution times. A *benchmark* is a program written specifically for this purpose. There are several well-known collections of benchmarks. One that is particularly interesting to computational scientists is LINPACK, which contains a set of linear algebra routines written in Fortran. MFLOPS ratings based on LINPACK performance are published regularly [8]. Two collections of a wider range of programs are SPEC (System Performance Evaluation Cooperative) and the Perfect Club, which is oriented toward parallel processing. Both include widely used programs such as a C compiler and a text formatter, not just small special purpose subroutines, and are useful for comparing systems such as high performance workstations that will be used for other jobs in addition to computational science modelling.

3 High Performance Computer Architecture

As described in Section 3.6 the performance of a computer system is defined by three factors. The time to execute a program is a function of the number of instructions to execute, the average number of clock cycles required per instruction, and the clock cycle time:

$$T = n_i \times \text{CPI} \times t_c$$

Lowering the clock cycle time is mostly a matter of engineering, through the use of more advanced materials or production techniques that allow the construction of smaller (and thus faster and more efficient) circuits. In this section we will survey several techniques for designing architectures that improve the other two factors.

The common thread that runs through all these techniques is *parallelism*, which is achieved by replicating basic components in the system. For example, an architect may use four adder/multiplier units instead of one inside the CPU, or connect two or more memories to the CPU in order to increase bandwidth, or connect two or more processors to one memory in order to increase the number of instructions executed per unit time, or even

replicate the entire computer (processor, memory, and I/O connections) in a network of machines that all work together on the same program. Parallelism has existed in the minds of computer architects from the time of Charles Babbage in the early 19th century, and has been manifested in a large number of machines in a variety of ways that might be classified in distinct levels [13]:

- *Job level parallelism*, the highest level of parallelism, is more of interest to administrators than individual users. What is most important from this point of view is that a lab or computer center execute as many jobs as possible in any given time period. This can be accomplished by purchasing more computer systems so more jobs are running at any one time, even though any one user's job will not run faster. Once again we see a distinction between throughput (number of jobs per day) and latency (the time to execute a program).
- *Program level parallelism* occurs when a single program is broken down into constituent parts. For example, the matrix product can be computed by breaking C into quadrants and having four processors compute each quadrant from the corresponding sections of A and B (also refer to the Chapter on Numerical Algebra). The entire product will be computed roughly four times faster since each processor can work independently of the others.
- *Instruction level parallelism* is mostly invisible to users, i.e. it is below the level of the architecture and in the domain of computer organization. Pipelines, introduced briefly in Section 3.6 and discussed in more detail below, are the most common way of implementing this type of parallelism.
- *Arithmetic and bit level parallelism* is the lowest level and is mainly of concern to designers of arithmetic-logic units inside the CPU. For example, a 64-bit sum can be computed by adding all 64 bits at once (the carry into the most significant bits can be predicted and computed almost as fast as the sum of any two bits), or for some reason the architect may decide to break the operation into 4-bit pieces and compute the entire sum in 16 cycles.

Job level parallelism is also exploited within a single computer by treating a job or several jobs as a collection of independent tasks. For example, several jobs may reside in memory at the same time with only one in execution at any given time. When that job requires some I/O services, such as a read from disc, the operation is initiated, the job requiring the service is suspended, and another job is put into execution state. At some future time, after the I/O operation has completed and the data is available, control will pass back to the original job and execution will continue. In this example, the CPU and the I/O system are functioning in parallel.

Parallelism at the program level is generally manifested in two ways: independent sections of a given program, and individual iterations of a loop where there are no dependencies between iterations. This type of parallelism may be exploited by multiple processors or

multiple functional units. For example, the following code segment calls for the calculation of n sums:

```
DO 10 I=1,N
  A(I) = B(I) + C(I)
10 CONTINUE
```

The sums are independent, i.e. the calculation of $b_i + c_i$ does not depend on $b_j + c_j$ for any $j < i$. That means they can be done in any order, and in particular a machine with n processors could do them all at the same time.

Obviously, more complex segments may also be treated in parallel depending on the sophistication of the architecture. This is the level of parallelism with which we will primarily be concerned in this book.

The next lower level of parallelism is at the instruction level, where individual instructions may be overlapped or a given instruction may be decomposed into suboperations with the suboperations overlapped. In the first case, for example, it is common to find a load instruction, which copies a value from memory to an internal CPU register, overlapped with an arithmetic instruction. The second situation is exemplified by the ubiquitous pipeline that has become the mainstay for arithmetic processing. In general, programmers need not concern themselves with this level of parallelism, since compilers are adept at reorganizing programs to exploit this form of parallelism. Nevertheless, one should keep in mind that the quality of compilers varies greatly from system to system and one may have to structure the code in particular ways to help the compiler make maximum use of the hardware. For example, as we will see below, Cray supercomputers are most efficient when vector lengths are 64 or smaller, and rearranging programs to operate on small segments of long vectors can improve performance. In addition, awareness of the internal structure of a computer is often necessary when analyzing the performance of a program.

A concept related to the level of parallelism is the *granularity* of parallel tasks. A large grain system is one in which the operations that run in parallel are fairly large, on the order of entire programs. Small grain parallel systems divide programs into very small pieces, in some cases only a few instructions. The example used above of a processor that calculates n sums in parallel is an example of very fine grain parallelism.

When designing a machine the architect must make a decision to use a relatively small number of powerful processors or a large number of simple processors to achieve the desired performance. The latter approach is often termed *massively parallel*.⁴ At one extreme are the systems built by Cray Research Inc. that consist of two to sixteen very powerful vector processors; at the other extreme are arrays of tens of thousands of very simple processors, exemplified by the CM-1 from Thinking Machines Corporation, which has up to 65,536 single-bit processors. The motivation for a small number of powerful processors is that they are simpler to interconnect and they lend themselves to an implementation of memory organizations that make the systems relatively easy to program. On the other

⁴What constitutes "massive" parallelism is not clearly defined. Most authors reserve the term for systems with 1000 or more individual processors.

hand such processors are very expensive to build, power and cool. Some architects have used commodity microprocessors which offer great economies of scale at the expense of more complex interconnection strategies. With the rapid increase in the power of microprocessors, arrays of a few hundred processors have the same theoretical peak performance of the fastest machines offered by Cray Research Inc.

This section of the book explores the design space of high performance machines, including single processor vector machines, parallel systems with a few powerful processors, and massively parallel architectures. Section 3.1 introduces a popular taxonomy of parallel systems. The next three sections cover major concepts of parallel systems organization, including pipelining, schemes for interconnecting processors and memories, and scalability. Section 3.5 is a survey of the major types of parallel machines, with an emphasis on systems that have been used in computational science. Finally, Section 3.6 presents models for analyzing the performance of parallel computer systems.

3.1 Flynn's Taxonomy

It is safe to say that as of this writing there is no completely satisfactory characterization of the different types of parallel systems. The most popular taxonomy was defined by Flynn in 1966 [9]. The classification is based on the notion of a stream of information. Two types of information flow into a processor: instructions and data. Conceptually these can be separated into two independent streams, whether or not the information actually arrives on a different set of wires.⁵ Flynn's taxonomy classifies machines according to whether they have one stream or more than one stream of each type (Figure 4). The four combinations are **SISD** (single instruction stream, single data stream), **SIMD** (single instruction stream, multiple data streams), **MISD** (multiple instruction streams, single data stream), and **MIMD** (multiple instruction streams, multiple data streams).

3.1.1 SISD Computers

Conventional single processor computers are classified as SISD systems. Each arithmetic instruction initiates an operation on a data item taken from a single stream of data elements. Historical supercomputers such as the Control Data Corporation 6600 and 7600 fit this category as do most contemporary microprocessors.

Vector processors such as the Cray-1 and its descendants are often classified as SIMD machines, although they are more properly regarded as SISD machines. Vector processors achieve their high performance by passing successive elements of vectors through separate pieces of hardware dedicated to independent phases of a complex operation. For example, in order to add two numbers such as 3.4×2^9 and 1.6×2^2 , the numbers must have the same exponent. The processor must shift the mantissa (and decrement the exponent) of one number until its exponent matches the exponent of the other number. In this example

⁵A processor is said to have a "Harvard architecture" if it has two separate memory channels, one for instructions and one for data.

3.4×2^9 is adjusted to 6.8×2^2 so it can be added to 1.6×2^2 , and the sum is 8.4×2^2 . A vector processor is specially constructed to feed a data stream into the processor at a high rate, so that as one part of the processor is adding the mantissas in the pair (a_i, b_i) another part of the processor is adjusting the exponents in (a_{i+1}, b_{i+1}) .

The ambiguity over the classification of vector machines depends on how one views the flow of data. A static "snapshot" of the processor during the processing of a vector would show several pieces of data being operated on at one time, and under this view one could say one instruction (a vector add) initiates several data operations (adjust exponents, add mantissas, etc.) and the machine might be classified SIMD. A more dynamic view shows that there is just one stream of data, and elements of this stream are passed sequentially through a single pipeline (which implements addition in this example). Another argument for not including vector machines in the SIMD category will be presented when we see how SIMD machines implement vector addition.

3.1.2 SIMD Computers

SIMD machines have one instruction processing unit, sometimes called a controller and indicated by a K in the PMS notation, and several data processing units, generally called *D-units* or *processing elements* (*PEs*). The first operational machine of this class was the ILLIAC-IV, a joint project by DARPA, Burroughs Corporation, and the University of Illinois Institute for Advanced Computation [5]. Later machines included the Distributed Array Processor (DAP) from the British corporation ICL, and the Goodyear MPP. Two recent machines, the Thinking Machines CM-1 and the MasPar MP-1, are discussed in detail in Section 3.1.2.

The control unit is responsible for fetching and interpreting instructions. When it encounters an arithmetic or other data processing instruction, it broadcasts the instruction to all PEs, which then all perform the same operation. For example, the instruction might be "add R3, R0." Each PE would add the contents of its own internal register R3 to its own R0. To allow for needed flexibility in implementing algorithms, a PE can be deactivated. Thus on each instruction, a PE is either idle, in which case it does nothing, or it is active, in which case it performs the same operation as all other active PEs. Each PE has its own memory for storing data. A memory reference instruction, for example "load R0, 100" directs each PE to load its internal register with the contents of memory location 100, meaning the 100th cell in its own local memory.

One of the advantages of this style of parallel machine organization is a savings in the amount of logic. Anywhere from 20% to 50% of the logic on a typical processor chip is devoted to control, namely to fetching, decoding, and scheduling instructions. The remainder is used for on-chip storage (registers and cache) and the logic required to implement the data processing (adders, multipliers, etc.). In an SIMD machine, only one control unit fetches and processes instructions, so more logic can be dedicated to arithmetic circuits and registers. For example, 32 PEs fit on one chip in the MasPar MP-1, and a 1024-processor system is built from 32 chips, all of which fit on a single board (the control unit occupies a separate

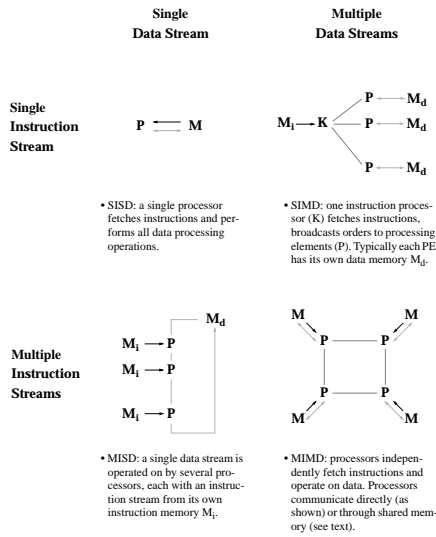


Figure 4: Flynn's Taxonomy

board).

Vector processing is performed on an SIMD machine by distributing elements of vectors across all data memories. For example, suppose we have two vectors, \mathbf{a} and \mathbf{b} , and a machine with 1024 PEs. We would store a_i in location 0 of memory i and b_i in location 1 of memory i . To add \mathbf{a} and \mathbf{b} , the machine would tell each PE to load the contents of location 0 into one register, the contents of location 1 into another register, add the two registers, and write the result. As long as the number of PEs is greater than the length of the vectors, vector processing on an SIMD machine is done in constant time, i.e. it does not depend on the length of the vectors. Vector operations on a pipelined SISD vector processor, however, take time that is a linear function of the length of the vectors.

3.1.3 MISD Computers

There are few machines in this category, none that have been commercially successful or had any impact on computational science. One type of system that fits the description of an MISD computer is a *systolic array*, which is a network of small computing elements connected in a regular grid. All the elements are controlled by a global clock. On each cycle, an element will read a piece of data from one of its neighbors, perform a simple operation (e.g. add the incoming element to a stored value), and prepare a value to be written to a neighbor on the next step.

One could make a case for pipelined vector processors fitting in this category, as well, since each step of the pipeline corresponds to a different operation being performed to the data as it flows past that stage in the pipe. There have been pipelined processors with programmable stages, i.e. the function that is applied at each location in the pipeline could vary, although the pipeline stage did not fetch its operation from a local control memory so it would be difficult to classify it as a "processor."

3.1.4 MIMD Computers

The category of MIMD machines is the most diverse of the four classifications in Flynn's taxonomy. It includes machines with processors and memory units specifically designed to be components of a parallel architecture, large scale parallel machines built from "off the shelf" microprocessors, small scale multiprocessors made by connecting four vector processors together, and a wide variety of other designs. With the continued improvement in network communication and the development of software packages that allow programs running on one machine to communicate with programs on other machines, users are even starting to use local networks of workstations as MIMD systems.

Computer systems with two or more independent processors have been available commercially for a long time. For example, the Burroughs Corporation sold dual processor versions of its B6700 systems in the 1970s. These were rarely, if ever, used to work on the same job, however. Multiprocessors of this era were intended to be used for job level parallelism, i.e. each would run a separate program. Parallel processing, in the sense of using more than one

processor in the execution of a single program, has been an active area in corporate and academic research labs since the early 1970s. The *c.mmp* and *cm** projects at Carnegie Mellon University used DEC PDP-11 microcomputers as processing elements and pioneered several important developments in parallel hardware and software. Commercial parallel processors started to become widely used in the mid 1980s. By the early 1990s these systems began to approach top of the line vector processors in computing power, and the trend for future high performance computing is clearly with parallel processing.

3.1.5 Other Taxonomies

In addition to the vacuous MISD category and the difficulty in classifying vector processors, there are other weaknesses of the Flynn taxonomy. In the MIMD category, all arrays of processors are lumped together regardless of how they are connected and how they view memory. Since these characteristics can have a dramatic effect on performance, it would be desirable if the taxonomy reflected those differences. Shore (@Shore) offered a very similar taxonomy, but expanded the SIMD category to four subcategories. He still did not distinguish the pipelined vector computer and he also did not provide for the completely independent array in the MIMD category. There have been other attempts to modify the Flynn taxonomy. For example in Hwang [14] the MIMD category is subdivided into shared memory systems, distributed memory systems and reconfigurable systems. Unfortunately, this mixes memory organization with communication organization, and although it is a useful distinction it is not very satisfactory as a basis for a taxonomy. Bell [3] divided the MIMD category into systems with shared memory and those without shared memory. One addition to the Flynn taxonomy that has become very popular is SPMD, which stands for Single Program / Multiple Data stream (see for example Karp [17]). In some sense it represents a style of computing rather than an architecture. Physically the system is an MIMD multiprocessor because there are several independent processors, each with its own data set and program memory. However, the same program is executed by each processor, and the processors are synchronized periodically. This is a much simpler way to approach an MIMD system than to have to manage many individual instruction streams. It also provides more flexibility than the SIMD system because different processors may be at different parts of the program at any time. By far the most ambitious attempt at a taxonomy is given by Hockney and Jesshope [13] where the motivation was to treat pipelined vector processors as a distinct architecture and to differentiate among the many multiprocessor possibilities. The notation resembles chemical notation for organic compounds and its complexity is beyond the scope of this discussion, but it does lead to a classification that provides a unique identifier for all of the systems that have been proposed or manufactured. However, that same complexity is the probable explanation for the lack of acceptance of the taxonomy.

3.2 Pipelines

A common analogy for a pipeline is the assembly line used in manufacturing. The end goal is to increase productivity – the number of instructions executed per second or the number

of cars built per day — by dividing a complex operation into pieces that can be performed in parallel. Separate "workers" implement successive steps along the assembly line, and when an item finishes one step it is passed down the line to next step.

Pipelines are used in two major areas in computer design: instruction processing and arithmetic operations. The following requirements must be satisfied in a pipelined system:

- A system is a candidate for pipelined implementation if it repeatedly executes a *basic function*.
- A basic function must be divisible into independent *stages* that have minimal overlap.
- The complexity of the stages should be roughly similar.

The number of stages is referred to as the *depth* of the pipeline. As an example of a pipeline, consider the floating point addition of two numbers of the form $m \times 2^e$. One possible breakdown of this function into stages is as follows [33]:

1. If $e_2 < e_1$ swap the operands. Find the difference in exponents $e_d = e_1 - e_2$.
2. Shift m_2 to the right by e_d bits.
3. Compute the mantissa of the sum by adding m_1 and m_2 . The exponent of the sum is e_1 .
4. Normalize the sum.

The extra complexity of a pipelined adder pays off when adding long sequences of numbers. Operations at each stage can be done on different pairs of inputs, e.g. one stage can be comparing the exponents in one pair of operands at the same time another stage is adding the mantissas of a different pair of operands.

A very important requirement for overlapping operations this way is that there be no resource conflicts, i.e. the operands must be independent. For example, suppose a program contains the two instructions

$R2 = R0 + R1$
 $R4 = R3 + R1$

Note that both instructions identify $R1$ as one of their inputs. There is a potential conflict in stages two and three of the adder pipeline because stage two might need to shift the mantissa of $R1$ at the same time stage three needs to add the mantissa of $R0$ to the mantissa of $R1$. The solution is to make copies of the operands, and pass the copies through the pipeline. Thus the CPU gives the adder copies of $R0$ and $R1$ when it starts the pipeline for the first pair, and the second stage gets these copies from the first stage along with a value of e_d .

There is another potential problem illustrated by this example. Suppose the second instruction is changed so one of its operands is $R2$, so the pair of instructions is:

R2 = R0 + R1
R4 = R2 + R1

Now the second instruction depends on the result of the first instruction. The CPU cannot send the second pair of operands to the pipelined adder until the result of the first addition exits the last stage of the pipeline. Interactions such as these lead to periods when pipeline stages are empty. These empty time slots are often called *bubbles*.

Figure 5 shows a type of diagram, known as a *Gantt chart*, that is commonly used to illustrate the operation of a pipeline. The horizontal axis represents time. There is one row for each stage of the pipeline. A line segment during cycle t_i means a stage is active in that cycle; a blank means the stage is inactive. The figure illustrates the pipelined floating point adder of the previous example in the case when two successive instructions can be overlapped and in the case where the second instruction must wait for the first to complete. In the case of overlapped instructions, note that in cycle t_1 the first stage is busy with the second instruction while the second stage is busy with the first instruction. In each successive cycle the two instructions are passed down the “assembly line” to the next stage. In the first case, the second sum is done after 6 cycles, but in the second case it is not finished until after the 10th cycle. The “bubble” in the pipeline is the 4-cycle dead period in each stage caused by delaying the second instruction.

It should be apparent that in the general case a pipeline of depth d can process n items in $n + d$ steps when there are no bubbles. Without a pipeline, each application of the basic function would require d cycles, and they would have to be executed sequentially, for a total time of $n \cdot d$ cycles. The speedup obtained by a full pipeline is thus

$$\frac{n \cdot d}{n + d}$$

When $n \gg d$ we can safely ignore the d in the denominator, so the asymptotic speedup, observed for large n , is a factor of d . For example, suppose we want to add 1000 pairs of numbers, e.g. when adding two 1000-element vectors. If it takes 5 cycles for each addition, a machine without a pipelined adder would require 5000 cycles. With our 5-stage pipelined adder, the last sum will appear after $1000 + 5$ cycles, so the pipeline is $5000/1005 = 4.97$ times faster. Providing a steady stream of independent operands that will keep a pipeline full is the distinguishing feature of a vector processor, which can initiate such a series of operations with a single instruction.

There are many possible sources of bubbles in pipelines. Dependencies between instructions are the main cause. For arithmetic pipelines, *data dependencies* arise when pairs of operations share inputs and outputs. Consider the following two add instructions:

z1 = x1 + y1
...
z2 = x2 + y2

The dependence illustrated previously is $x2 = z1$, e.g. both operands are the register R2. Other dependencies are $z1 = z2$ (both instructions write to the same register) and $x1 = z2$

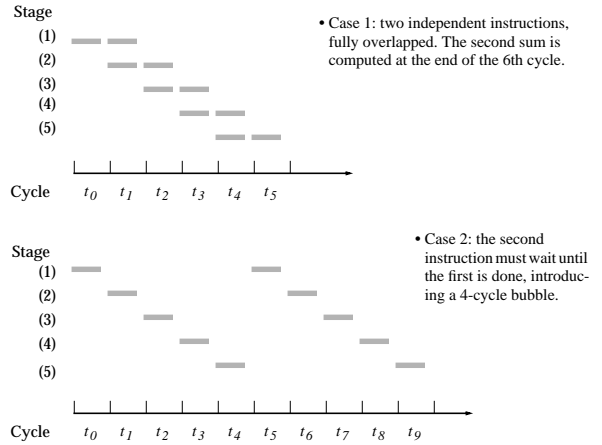


Figure 5: Gantt Charts for Pipelined Floating Point Adder

(the output from the second instruction may overwrite the register before the first has had a chance to read the old value; an unlikely occurrence in a vector machine, but a situation that must be taken into account). Note that the instructions do not have to be consecutive, i.e. there could be intervening instructions. A compiler that checks for dependencies and possibly reorders instructions has to “look ahead” in the code by an amount equal to the depth of the pipeline used in the first instruction.

Instruction pipelines, used to speed up the fetch-decode-execute cycle, are also susceptible to bubbles. The most common case here is caused by branch or loop instructions, which are *control dependencies*. If the pipeline is “looking ahead” and fetching instructions it thinks the machine will want to execute, but in fact the machine branches to another location, a bubble is introduced while the fetch stage goes to get the instructions at the new location.

Pipelines have been widely used in high performance machines for many years. The CDC 6600 is a classic example of a complex instruction pipeline, using a circuit known as a “scoreboard” to detect data dependencies between instructions and instruction buffers to implement pipelining in the fetch-decode logic. The Cray-1 was a very successful early supercomputer in which every data processing unit was pipelined. Until the late 1980s pipelining was one of the attributes that separated “mainframes” and supercomputers from microprocessors, but with the advance of VLSI technology microprocessors now have room on chip for complex control circuitry. Most now have pipelined data processing units and complex instruction scheduling logic that rivals that of the CDC 6600, an early pipelined processor known for its innovative instruction processing.

3.3 Memory Organizations

So far the discussion of high performance computing has concentrated on increasing the amount of processing power in a system, either through parallelism, which seeks to increase the number of instructions that can be executed in a time period, or through pipelining, which improves the instruction throughput. Another, equally important, aspect of high performance computing is the organization of the memory system. No matter how fast one makes the processing unit, if the memory cannot keep up and provide instructions and data at a sufficient rate there will be no improvement in performance. The main problem that needs to be overcome in matching memory response to processor speed is the memory cycle time, defined in section 2.2 to be the time between two successive memory operations. Processor cycle times are typically much shorter than memory cycle times. When a processor initiates a memory transfer at time t_0 , the memory will be “busy” until $t_0 + t_c$, where t_c is the memory cycle time. During this period no other device — I/O controller, other processors, or even the processor that makes the request — can use the memory since it will be busy responding to the request.

Solutions to the memory access problem have led to a dichotomy in parallel systems. In one type of system, known as a *shared memory* system, there is one large virtual memory, and all processors have equal access to data and instructions in this memory. The other type of system is a *distributed memory*, in which each processor has a *local memory* that is not

accessible from any other processor.

The difference between shared or distributed memory is a difference in the structure of virtual memory, i.e. the memory as seen from the perspective of a processor. Physically, almost every memory system is partitioned into separate components that can be accessed independently. What distinguishes a shared memory from a distributed memory is how the memory subsystem interprets an address generated by a processor. As an example, suppose a processor executes the instruction `load R0, i`, which means “load register R0 with the contents of memory location i ” (denoted `Mem[i]`). The question is, what does i mean? In a shared memory system, i is a global address, and `Mem[i]` to one processor is the same memory cell as `Mem[i]` to another processor. If both processors execute this instruction at the same time they will both load the same information into their R0 registers. In a distributed memory system, i is a local address. If two processors both execute `load R0, i` they may end up with different values in their R0 registers since `Mem[i]` designates two different memory cells, one in the local memory of each processor.

The distinction between shared memory and distributed memory is an important one for programmers because it determines how different parts of a parallel program will communicate. In a shared memory system it is only necessary to build a data structure in memory and pass references to the data structure to parallel subroutines. For example, a matrix multiplication routine that breaks matrices into quadrants only needs to pass the indices of each quadrant to the parallel subroutines. A distributed memory machine on the other hand must create copies of shared data in each local memory. These copies are created by sending a *message* containing the data to another processor. In the matrix multiplication example, the controlling process would have to send messages to three other processors. Each message would contain the submatrices required to compute one quadrant of the result. A drawback to this memory organization is that these messages might have to be quite large; in this example, half of each input matrix needs to be sent to each parallel subroutine.

In this section we will explore the range of techniques used to connect processors to memories in high performance computers and how these techniques affect programmers. The first section is on interleaved memory, a method long used in vector processors to provide successive vector elements at a rate that matches the cycle time in the pipelined data processing units. The next two sections deal with shared memory and distributed memory organizations for parallel systems.

3.3.1 Interleaved Memory

In an interleaved memory, the memory is divided into a set of *banks*. An interleaved memory with n banks is said to be *n-way interleaved*. One way of allocating virtual addresses to memory modules is to divide the memory space (the set of all possible addresses a processor can generate) into contiguous blocks. If there are n banks, memory location i would reside in bank number i/n (ignoring remainders). In an interleaved memory, however, consecutive addresses reside in different banks: memory location i is in bank number $i \bmod n$. For example, suppose there are 4 banks, each containing 256 bytes. The block-oriented scheme

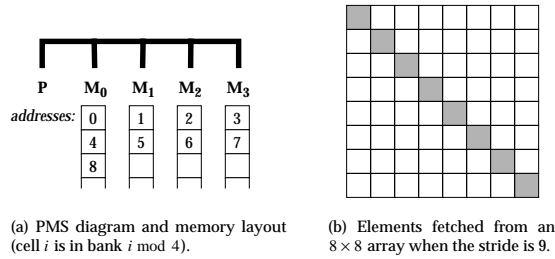


Figure 6: Interleaved Memory

would assign virtual addresses $0 \dots 255$ to the first bank, $256 \dots 511$ to the second bank, and so on. The interleaved scheme would assign addresses $0, 4, 8, \dots$ to the first bank, $1, 5, 9, \dots$ to the second bank, etc. (Figure 6).

However the memory space is split up among the banks, as long as requests are sent to two different banks they can be handled simultaneously. The processor can request a transfer from location i on one cycle, and on the next cycle request information from location j . If i and j are in different banks, the information will be returned on successive cycles. Note that the latency of the request, i.e. the number of cycles a processor has to wait before receiving the contents of location i , is not affected. However the bandwidth is improved; if there are enough banks the memory system can potentially send information at a rate of one word per processor cycle, regardless of what the memory cycle time is.

The decision to allocate addresses as contiguous blocks or in interleaved fashion depends on how one expects information to be accessed. Programs are compiled so instructions reside in successive addresses, so there is a high probability that after a processor executes the instruction at location i it will execute the instruction at $i + 1$ (Section 2.2). Compilers can also allocate vector elements to successive addresses, so operations on entire vectors can take advantage of interleaving. For these reasons, vector processors universally have some form of interleaved memory. However, shared memory multiprocessors use the block-oriented scheme since memory referencing patterns in an MIMD system are quite different. There the goal is to connect a processor to a single memory and use as much information as possible from that memory before switching to another memory.

Systems often provide some flexibility in fetching vector elements. In some systems it is possible to load every n^{th} element, for example when fetching elements of a vector v that is

stored in consecutive memory cells with $n = 4$ the memory would return v_0, v_4, v_8 . The interval between elements is known as the *stride*. One interesting use of this feature is in accessing matrices. If the stride is set to one more than the number of rows, a single memory request will return the diagonal elements (assuming column major layout and the columns are stored contiguously). Using a stride may cancel any benefits of interleaving if programmers are not careful. In an extreme case, setting the stride to the degree of interleaving means every item is fetched from the same bank and the time between successive elements will be the memory cycle time.

3.3.2 Shared Memory

A straightforward way to connect several processors together to build a multiprocessor is shown in Figure 7. The physical connections are quite simple. Most bus structures allow an arbitrary (but not too large) number of devices to communicate over the bus. Bus protocols were initially designed to allow a single processor and one or more disk or tape controllers to communicate with memory. If the I/O controllers are replaced by processors, one has a small single-bus multiprocessor.

The problem with this design is that processors must contend for access to the bus. If a processor P_j is fetching an instruction, all other processors $P_{j'} \neq j$ must wait until the bus is free. If there are only two processors they can perform close to their maximum rate since the bus can alternate between them: as one processor is decoding and executing an instruction, the other can be using the bus to fetch its next instruction. However, when a third processor is added performance begins to degrade. Usually by the time 10 processors are connected to the bus the performance curve has flattened out so that adding an 11th processor will not increase performance at all. The bottom line is the fact that the memory and bus have a fixed bandwidth, determined by a combination of the cycle time of the memory and the bus protocol, and in a single-bus multiprocessor this bandwidth is divided among several processors. If the processor cycle time is very slow compared to the memory cycle, a fairly large number of processors can be accommodated by this plan, but in fact processor cycles are usually much faster than memory cycles so this scheme is not widely used.

A slight modification to this design will improve performance, but it cannot indefinitely postpone the flattening of the performance curve. If each processor has its own local cache, there is a high probability ($p > 0.9$) that the instruction or data it wants is in the local cache. A reasonable cache hit rate will greatly reduce the number of accesses a processor makes and thus improve overall efficiency. The “knee” of the performance curve, which identifies a point where it is still cost-effective to add processors, can now be around 20 processors, and the curve will not flatten out until around 30 processors.

Giving each processor its own cache introduces a difficulty known as the *cache coherency problem*. In its simplest form, the problem may be exemplified by the following scenario. Suppose two processors use data item A , so A ends up in the cache of both processors. Next suppose processor 1 performs a calculation that changes A . When it is done, the new value

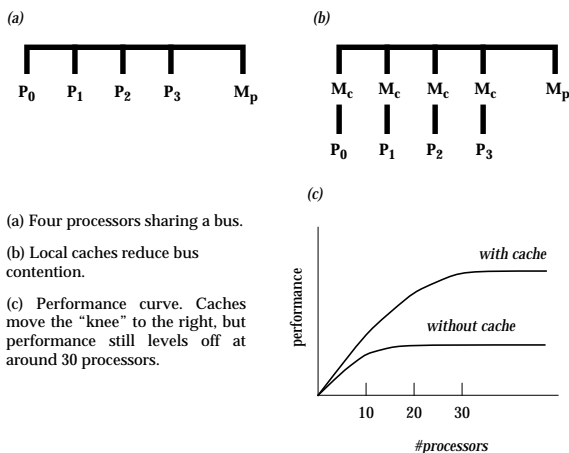


Figure 7: Single Bus Multiprocessor

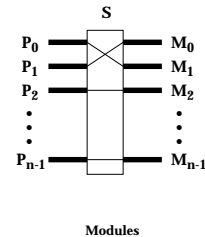


Figure 8: Shared Memory Multiprocessor with Discrete Memory Modules

of A is written out to main memory.⁶ Processor 2 at a later time needs to fetch A . However, since A was already in its cache, it will use the cached value and not the newly updated value calculated by processor 1. Maintaining a consistent version of shared data requires providing new versions of the cached data to each processor whenever one of the processors updates its copy.

The multiprocessors produced by Sequent, Inc. are classic examples of machines of this type. Their first machine, the Balance 8000, was intended to compete with the DEC VAX 780, a popular minicomputer at that time. A 2-processor configuration gave slightly less performance than the VAX, but the next larger configuration, with four processors, was faster. The operating system was a modified version of Unix. There was a single global task queue, and each processor could fetch a task from the queue, execute it until it blocks or times out, and return it to the queue. Thus the system implemented a form of job level parallelism. Sequent also provided a library of procedures that allowed users to write parallel programs, and the machine became a popular testbed for parallel languages and algorithms. The current machines, in the Symmetry series, are widely used for on-line transaction processing.

Programming a shared memory machine is fairly straightforward. Programming constructs such as semaphores, fork-join, and monitors, which were developed for communication and synchronization of parallel processes in operating systems and other concurrent programming applications, have been adapted for parallel processing. The implementation of the basic synchronization primitives from which these constructs are built is more complex in a parallel system, but this complexity is hidden from users. For example, the bus in the Sequent Symmetry has provisions for implementing a pool of semaphores so that processes are guaranteed to gain exclusive access to shared structures.

Another way of building a shared memory multiprocessor is shown in Figure 8. In these

⁶Most single-bus multiprocessors use a design known as *write-through cache* in which values written to memory are sent simultaneously to the cache and to the main memory.

designs, the bus has been replaced by a switch that routes requests from a processor to one of several different memory modules. Even though there are several physical memories, there is one large virtual address space. The advantage of this organization is based on the fact the switch can handle multiple requests in parallel. Each processor can be paired up with a memory, and each can then run at full speed as it accesses the memory it is currently connected to. Contention still occurs, though. If two processors make requests of the same memory module only one will be given access and the other will be blocked. Several machines with this design will be discussed in the survey of MIMD machines following the section on interconnection topology, which introduces concepts that will explain various switch designs.

3.3.3 Distributed Memory

In a distributed memory system the memory is associated with individual processors and a processor is only able to address its own memory. Some authors refer to this type of system as a *multicomputer*, reflecting the fact that the building blocks in the system are themselves small computer systems complete with processor and memory.

There are several benefits of this organization. First, there is no bus or switch contention. Each processor can utilize the full bandwidth to its own local memory without interference from other processors. Second, the lack of a common bus means there is no inherent limit to the number of processors; the size of the system is now constrained only by the network used to connect processors to each other. Third, there are no cache coherency problems. Each processor is in charge of its own data, and it does not have to worry about putting copies of it in its own local cache and having another processor reference the original.

The major drawback in the distributed memory design is that interprocessor communication is more difficult. If a processor requires data from another processor's memory, it must exchange messages with the other processor. This introduces two sources of overhead: it takes time to construct and send a message from one processor to another, and a receiving processor must be interrupted in order to deal with messages from other processors.

Programming on a distributed memory machine is a matter of organizing a program as a set of independent tasks that communicate with each other via messages. In addition, programmers must be aware of where data is stored, which introduces the concept of locality in parallel algorithm design. An algorithm that allows data to be partitioned into discrete units and then runs with minimal communication between units will be more efficient than an algorithm that requires random access to global structures.

Semaphores, monitors, and other concurrent programming techniques are not directly applicable on distributed memory machines, but they can be implemented by a layered software approach. User code can invoke a semaphore, for example, which is itself implemented by passing a message to the node that "owns" the semaphore. This approach is not very efficient, however, and it has the drawback of *nonuniform memory access*, i.e. the latency of a memory request, in this case reading the value of a semaphore, is proportional to the distance between the processor making the request and the memory where the value is stored.

Which programming style is easier — shared memory with semaphores, etc. or dis-

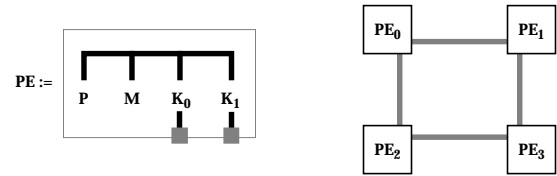


Figure 9: Distributed Memory Parallel Processor

tributed memory with message passing — is often a matter of personal preference. The message passing style fits very well with the object oriented programming methodology, and if a program is already organized in terms of objects it may be quite easy to adapt it for a distributed memory system. When faced with a decision of whether to implement a program in shared memory or distributed memory the outcome is usually based on the amount of information that must be shared by parallel tasks. Whatever information is shared among tasks must be copied from one node to another via messages in a distributed memory system, and this overhead may reduce efficiency to the point where a shared memory system is preferred.

A PMS diagram of a simple distributed memory parallel processor is shown in Figure 9. On the left is the diagram of a single node, often called a *processing element*, or PE. The organization of a PE explains how messages are passed from one PE to another. As far as any one processor is concerned, the other processors are simply I/O devices. To send a message to another PE, a processor copies information into a data block in its local memory and then tells its local controller to transfer the information to an external device, much the same way a disk controller in a microcomputer would write a block on a disk drive. In this case, however, the block of data is transferred over the interconnection network to an I/O controller in the receiving node. That controller finds room for the incoming message in its local memory and then notifies the processor that a message has arrived.

3.4 Topology

A major consideration in the design of parallel systems is the set of pathways over which the processors, memories, and switches communicate with each other. These connections define the *interconnection network*, or *topology*, of the machine. Attributes of the topology determine how processors will share data and at what cost.

The following discussion of the properties of interconnection networks is based on a collection of nodes that communicate via links. In an actual system the nodes can be either processors, memories, or switches. Unless otherwise noted the links will always be point-to-

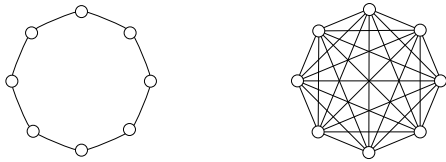


Figure 10: Ring vs. Fully Connected Network

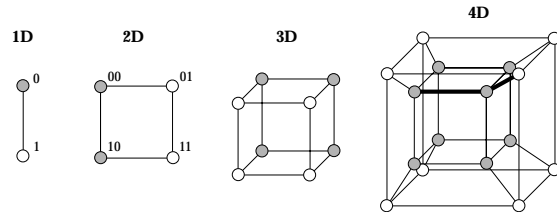
point data paths, i.e. not buses that are shared by several nodes. The properties discussed here apply equally to MIMD and SIMD machines, or to shared memory or distributed memory architectures. Examples of most of the topologies will be given in the survey of high performance systems (Section 3.5).

Two nodes are *neighbors* if there is a link connecting them. The *degree* of a node is defined to be the number of its neighbors. Figure 10 shows two common topologies, a ring and a fully connected network, each with eight nodes. Each node in the ring is connected to only two other nodes, while each node in the fully connected network is linked to every other node. In practice the degree of a topology has an effect on cost, since the more links a node has the more logic it takes to implement the connections.

When a node is not connected to every other node, messages may have to go through intervening nodes to reach their final destination. The *diameter* of a network is the longest path between any two nodes. Again the ring and fully connected network show two extremes. A ring of n nodes has diameter $n/2$, but a fully connected network has a fixed diameter (1) no matter how many nodes there are.

The diameter of a ring grows as more nodes are added, but the diameter of a fully connected network remains the same. On the other hand, a ring can expand indefinitely without changing the degree, but each time a new node is added to a fully connected network a link has to be added to each existing node. *Scalability* refers to the increase in the complexity of communication as more nodes are added. In a highly scalable topology more nodes can be added without severely increasing the amount of logic required to implement the topology and without increasing the diameter.

A scalable topology that has been used in several parallel processors is the hypercube, shown in Figure 11. A line connecting two nodes defines a 1-dimensional "cube." A square with four nodes is a 2-dimensional cube, and a 3D cube has eight nodes. This pattern reveals a rule for constructing an n -dimensional cube: begin with an $(n-1)$ -dimensional cube, make



To construct an n -dimensional cube, copy an $(n-1)$ -dimensional cube, then connect corresponding nodes in the original and the copy. In these figures nodes from the original are colored gray.

Figure 11: Hypercubes

an identical copy, and add links from each node in the original to the corresponding node in the copy. Doubling the number of nodes in a hypercube increases the degree by only 1 link per node, and likewise increases the diameter by only 1 path. It is left as an exercise to prove that an n -dimensional hypercube has 2^n nodes, diameter n , and degree n .

Communication in a hypercube is based on the binary representation of node IDs. The nodes are numbered so that two nodes are adjacent if and only if the binary representations of their IDs differ by one bit. For example, nodes 0110 and 0100 are immediate neighbors but 0110 and 0101 are not. An easy way to label nodes is to assign node IDs as the cube is constructed. When you copy an $(n-1)$ -dimensional cube, make sure the corresponding nodes in the two copies have the same IDs. Then extend all the IDs by one bit. Append a 0 to the IDs of nodes in the original cube, and append a 1 to the IDs of nodes in the copy. As an example the nodes in the 1D and 2D cubes in Figure 11 are labeled according to this scheme; the labeling of the 3D and 4D cubes is left for an exercise.

Node IDs are the basis for a simple algorithm for routing information in a hypercube. An n -dimensional cube will have n -bit node IDs. Sending a message from node A to node B can be done in n cycles, where on each cycle a node will either hold a message or forward it along one of its links. On cycle i the node that currently holds the message will compare bit i of its own ID with bit i of the destination ID. If the bits match, the node holds the message. If they don't match, it forwards the message along dimension i , where dimension i is the dimension that was added in the i^{th} step of the construction of the cube (i.e. it is

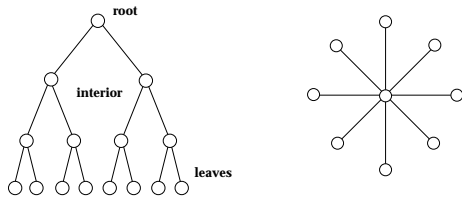


Figure 12: Tree and Star

the same “direction” at all nodes). As an example, the path from node 2 to node 7 in a 4D cube is marked with a heavy gray line in Figure 11.

Another desirable property of interconnection networks is *node symmetry*. A node symmetric network has no distinguished node, that is, the “view” of the rest of the network is the same from any node. Rings, fully connected networks, and hypercubes are all node symmetric. Trees and stars, shown in Figure 12, are not. A tree has three different types of nodes, namely a root node, interior nodes, and leaf nodes, each with a different degree. A star has a distinguished node in the center which is connected to every other node. When a topology is node asymmetric a distinguished node can become a communications bottleneck.

A more formal definition of a communication bottleneck is based on a property known as the *bisection width*, which is the minimum number of links that must be cut in order to divide the topology into two independent networks of the same size (plus or minus one node). The bisection width of a tree is 1, since if either link connected to the root is removed the tree is split into two subtrees. The *bisection bandwidth* of a parallel system is the communication bandwidth across the links that are cut in defining the bisection width. This bandwidth is useful in defining worst-case performance of algorithms on a particular network, since it is related to the cost of moving data from one side of the system to the other.

Another common topology is a planar (2D) *mesh*, shown in Figure 13. This network is basically a matrix of nodes, each with connections to its nearest neighbors. Meshes usually have “wraparound” connections, e.g. the node at the top of the grid has an “up” link that connects to the node at the bottom of the grid. If you visualize only north-south links in a rectangular mesh, you can see these links turn the 2D mesh into a 3D cylinder. Now if the east-west links are added, it connects the ends of the cylinder to form a toroidal solid. Thus a mesh topology with wraparound connections is often referred to as a *torus*. In many

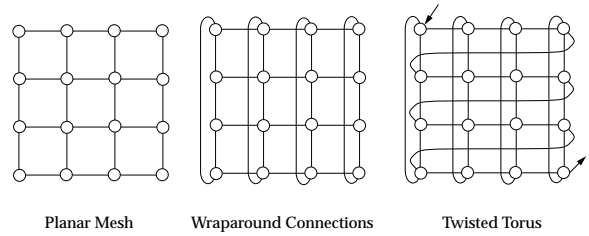


Figure 13: Mesh Topologies

systems the wraparound connections are skewed by one or more rows (or columns, or both); in this case the topology is known as a *twisted torus*. Note that a path that starts in the northwest corner of a twisted torus and heads continually east will visit every node exactly once before returning to the northwest corner.

The two final interconnection networks introduced in this section are examples of *multistage networks*. Systems built with these topologies have processors on one edge of the network, memories or processors on another edge, and a series of switching elements at the interior nodes. In order to send information from one edge to another, the interior switches are configured to form a path that connects nodes on the edges. The information then goes from the sending node, through one or more switches, and out to the receiving node. The size and number of interior nodes contributes to the path length for each communication, and there is often a “setup time” involved when a message arrives at an interior node and the switch decides how to configure itself in order to pass the message through.

The first example of a multistage network is the crossbar switch Figure 14. In a typical application there will be a column of processors on the left edge and a row of memories on the bottom edge. The switch configures itself dynamically to connect a processor to a memory module. As long as each processor wants to communicate with a different memory there will be no contention. If two or more processors need to access the same memory, however, one will be blocked until the switch reconfigures itself. A crossbar has a short diameter — information needs to pass through only one switching element on a path from one edge to another — but poor scalability. If there are n processors and a like number of memories there are n^2 interior switches. Adding another processor and memory means adding another $2n - 1$ interior nodes.

A *banyan network* is a multistage switching network that has the same number of inputs

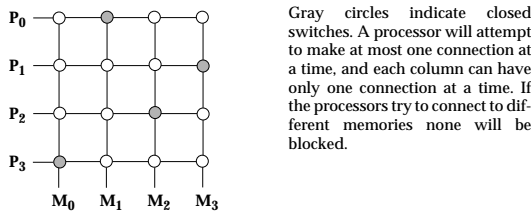


Figure 14: Crossbar Switch

as outputs and interior nodes that are $m \times m$ switches. Examples of banyan networks are *butterfly networks* and *omega networks*, which are both built from 2×2 switches. The diameter of a butterfly is $\log_2 n$, where n is the number of inputs and outputs, and there are $O(n \cdot \log_2 n)$ switches, so these networks scale more efficiently than a crossbar (Figure 15). The 2×2 switch in a butterfly can be configured in one of two states (Figure 15). One configuration connects input 0 to output 0 and input 1 to output 1. The other configuration flips the outputs, so input 0 connects to output 1 and input 1 connects to output 0. The switching network uses the binary representation of the destination address in order to construct a path from input to output. The switch at stage i in the network uses bit i to determine how to configure itself: if the bit is 0, the request should go through the top output, and if it is 1 it should go through the bottom output. For example, suppose a processor needs to fetch information from memory M_5 . The binary representation of 5 is 101. The first switch will pass the request out its bottom output, the second switch will pass the request out its top output, and the last switch will pass the request out its bottom output. Note that this pattern of connections (top-bottom-top) works no matter which processor generates the request. Whether a switch configures itself in the straight-through or flipped configuration depends on which input the request comes from. For example, if the request comes from the top input and should be routed out the top output, then the switch will go into the straight-through configuration, but if the request comes from the top input and should go out the bottom the switch will use the flipped configuration.

As is the case with the crossbar switch, there are configurations of the butterfly that will allow each processor to connect to a different memory so all processors can be active and no requests are blocked. However, the butterfly is not as flexible as the crossbar, since combinations of requests that are nonblocking in the crossbar are blocking in the butterfly.

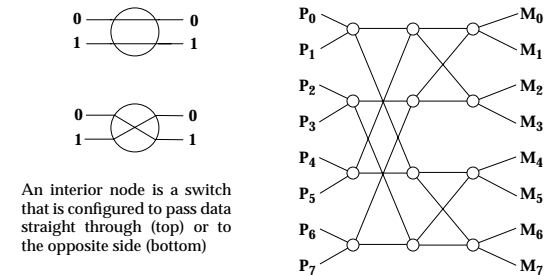


Figure 15: Butterfly Network

For example, if the first switch in the first column is in the straight-through configuration because processor P_0 is making a request to memory M_2 , processor P_1 is constrained to communicate with memories 4 through 7 (100_2 through 111_2). With a crossbar P_1 would be allowed to connect to M_0 , M_1 , or M_3 without blocking.

Crossbar and butterfly switches have both been used to implement shared memory multiprocessors. Even though there are independent memory modules, there is a single memory space, i.e. an address i generated by one processor refers to the same cell as an address i generated by any other processor. Addresses are not interleaved, though. Instead the memory space is divided into contiguous blocks of equal size. For example, suppose there are 4 memory units and the address space has $2^{10} = 1024$ words. M_0 would hold addresses 0 to 255, M_1 would have 256 to 511, and so on.

Three important attributes of an interconnection network are the *timing strategy*, *control strategy*, and *switching strategy* [4]. The two alternatives for control are a single central controller or a distributed control system in which routing strategies are implemented in each node. Message routing based on node IDs in hypercubes and butterfly switches are examples of distributed control, since each node decides for itself how to reroute incoming messages. A centralized strategy would work well in a star network: messages from outer nodes must pass through the center, which would then decide how to forward the message. Synchronous control techniques are characterized by a global clock that broadcasts clock signals to all devices in a system so that the entire system operates in a lock-step fash-

ion. Asynchronous techniques do not utilize a single global clock, but rather distribute the control function throughout the system, often utilizing many individual clocks for timing. Control and coordination of the various parts of the system are accomplished via some form of communication or “hand shaking.” Thus the interconnection network can operate synchronously off of a global clock or it may have distributed control down to the level of the individual switches. The advantage of a single global clock for control is simplicity in both the hardware and the software; the advantage of distributed control is expandability and flexibility. Synchronous and asynchronous timing strategies are a fundamental characteristic of computing systems in general. The SIMD systems discussed previously normally operate synchronously with a global clock while the MIMD systems function asynchronously with a clock in each PE.

Switching strategy is the other important characteristic of interconnection networks. The two most popular techniques are *packet switching* and *circuit switching*. In packet switching, a message is broken into small packets which are transmitted through the network in a “store and forward” mode. A packet traverses one link, where the receiving node will examine it and decide what to do. It may have to store the packet for a while before forwarding it toward its final destination, e.g. there may be other packets waiting to go out on that link. It is also possible that packets will traverse different sets of links on their route from source to destination. Packets may experience delays at each switching point depending on the traffic in the network. The circuit switching technique establishes a complete path between the source and the destination and then starts transferring information along the path. The circuit is kept open until the entire message has been transmitted. We will see examples of both strategies in the section on MIMD systems.

3.5 Survey of High Performance Architectures

3.5.1 Vector Processors

A vector processor is a processor that can operate on entire vectors with one instruction, i.e. the operands of some instructions specify complete vectors. For example, consider the following add instruction:

```
C = A + B
```

In both scalar and vector machines this means “add the contents of **A** to the contents of **B** and put the sum in **C**.” In a scalar machine the operands are numbers, but in vector processors the operands are vectors and the instruction directs the machine to compute the pairwise sum of each pair of vector elements. A processor register, usually called the vector length register, tells the processor how many individual additions to perform when it adds the vectors.

A vectorizing compiler is a compiler that will try to recognize when loops can be transformed into single vector instructions. For example, the following loop can be executed by a single instruction on a vector processor:

50

```
DO 10 I=1,N
  A(I) = B(I) + C(I)
10 CONTINUE
```

This code would be translated into an instruction that would set the vector length to **N** followed by a vector add instruction.

The use of vector instructions pays off in two different ways. First, the machine has to fetch and decode far fewer instructions, so the control unit overhead is greatly reduced and the memory bandwidth necessary to perform this sequence of operations is reduced a corresponding amount. The second payoff, equally important, is that the instruction provides the processor with a regular source of data. When the vector instruction is initiated, the machine knows it will have to fetch *n* pairs of operands which are arranged in a regular pattern in memory. Thus the processor can tell the memory system to start sending those pairs. With an interleaved memory, the pairs will arrive at a rate of one per cycle, at which point they can be routed directly to a pipelined data unit for processing. Without an interleaved memory or some other way of providing operands at a high rate the advantages of processing an entire vector with a single instruction would be greatly reduced.

A key division of vector processors arises from the way the instructions access their operands. In the *memory to memory* organization the operands are fetched from memory and routed directly to the functional unit. Results are streamed back out to memory as the operation proceeds. In the *register to register* organization operands are first loaded into a set of *vector registers*, each of which can hold a segment of a register, for example 64 elements. The vector operation then proceeds by fetching the operands from the vector registers and returning the results to a vector register.

The advantage of memory to memory machines is the ability to process very long vectors, whereas register to register machines must break long vectors into fixed length segments. Unfortunately, this flexibility is offset by a relatively large overhead known as the *startup time*, which is the time between the initialization of the instruction and the time the first result emerges from the pipeline. The long startup time on a memory to memory machine is a function of memory latency, which is longer than the time it takes to access a value in an internal register. Once the pipeline is full, however, a result is produced every cycle or perhaps every other cycle. Thus a performance model for a vector processor is of the form

$$T = s + aN$$

where *s* is the startup time, *N* is the length of the vector and *a* is an instruction dependent constant, usually 1/2, 1 or 2.

Examples of this type of architecture include the Texas Instruments Inc. Advanced Scientific Computer and a family of machines built by Control Data Corp. known first as the Cyber 200 series and later the ETA-10 when Control Data Corp. founded a separate company known as ETA Systems Inc. These machines appeared in the mid 1970s after a long development cycle that left them with dated technology and disappeared in the mid 1980s. For a thorough discussion of their characteristics, see Hockney and Jesshope [13]. One

of the major reasons for their demise was the large startup time, which was on the order of 100 processor cycles. This meant that short vector operations were very inefficient, and even for vectors of length 100 the machines were delivering only about half their maximum performance. In a later section we will see how this vector length that yields half of peak performance is used to characterize vector computers.

In the register to register machines the vectors have a relatively short length, 64 in the case of the Cray family, but the startup time is far less than on the memory to memory machines. Thus these machines are much more efficient for operations involving short vectors, but for long vector operations the vector registers must be loaded with each segment before the operation can continue. Register to register machines now dominate the vector computer market, with a number of offerings from Cray Research Inc., including the Y-MP and the C-90. The approach is also the basis for machines from Fujitsu, Hitachi and NEC. Clock cycles on modern vector processors range from 2.5ns (NEC SX-3) to 4.2ns (Cray C90), and single processor performance on LINPACK benchmarks is in the range of 1000 to 2000 MFLOPS (1 to 2 GFLOPS).

The basic processor architecture of the Cray supercomputers has changed little since the Cray-1 was introduced in 1976 [28]. There are 8 vector registers, named V0 through V7, which each hold 64 64-bit words. There are also 8 scalar registers, which hold single 64-bit words, and 8 address registers (for pointers) that have 20-bit words. Instead of a cache, these machines have a set of backup registers for the scalar and address registers; transfer to and from the backup registers is done under program control, rather than by lower level hardware using dynamic memory referencing patterns.

The original Cray-1 had 12 pipelined data processing units; newer Cray systems have 14. There are separate pipelines for addition, multiplication, computing reciprocals (to divide *X* by *Y*; a Cray computes *X* · (1/*Y*)), and logical operations. The cycle time of the data processing pipelines is carefully matched to the memory cycle times. The memory system delivers one value per clock cycle through the use of 4-way interleaved memory.

An interesting feature introduced in the Cray computers is the notion of *vector chaining*. Consider the following two vector instructions:

```
V2 = V0 * V1
V4 = V2 + V3
```

The output of the first instruction is one of the operands of the second instruction. Recall that since these are vector instructions, the first instruction will route up to 64 pairs of numbers to a pipelined multiplier. About midway through the execution of this instruction, the machine will be in an interesting state: the first few elements of **V2** will contain recently computed products; the products that will eventually go into the next elements of **V2** are still in the multiplier pipeline; and the remainder of the operands are still in **V0** and **V1**, waiting to be fetched and routed to the pipeline. This situation is shown in Figure 16, where the operands from **V0** and **V1** that are currently in the multiplier pipeline are indicated by gray cells. At this point, the system is fetching **V0[k]** and **V1[k]** to route them to the first stage of the pipeline and **V2[j]** is just leaving the pipeline. Vector chaining relies on the

52

path marked with an asterisk. While **V2[j]** is being stored in the vector register, it is also routed directly to the pipelined adder, where it is matched with **V3[j]**. As the figure shows, the second instruction can begin even before the first finished, and while both are executing the machine is producing two results per cycle (**V4[i]** and **V2[j]**) instead of just one.

Without vector chaining, the peak performance of the Cray-1 would have been 80 MFLOPS (one full pipeline producing a result every 12.5ns, or 80,000,000 results per second). With three pipelines chained together, there is a very short burst of time where all three are producing results, for a theoretical peak performance of 240 MFLOPS.⁷ In principle vector chaining could be implemented in a memory-to-memory vector processor, but it would require much higher memory bandwidth to do so. Without chaining, three “channels” must be used to fetch two input operand streams and store one result stream; with chaining, five channels would be needed for three inputs and two outputs. Thus the ability to chain operations together to double performance gave register- to-register designs another competitive edge over memory-to- memory designs.

3.5.2 Superscalar Processors

The evolution of microprocessors has reached the point where architectural concepts pioneered in vector processors and mainframe computers of the 1970s (most notably the CDC-6600 and Cray-1) are starting to appear in RISC processors. Early RISC machines were very simple single-chip processors. As VLSI technology improved more room became available on the chip. Rather than increase the complexity of the architecture, most designers decided to use this room on techniques to improve the execution of their current architecture. The two principle techniques are on-chip caches and instruction pipelines.

The latest step in this evolutionary process is the *superscalar processor*. The name means these processors are scalar processors that are capable of executing more than one instruction in each cycle. The keys to superscalar execution are an instruction fetching unit that can fetch more than one instruction at a time from cache; instruction decoding logic that can decide when instructions are independent and thus executed simultaneously; and sufficient execution units to be able to process several instructions at one time. Note that the execution units may be pipelined, e.g. they may be floating point adders or multipliers, in which case the cycle time for each stage matches the cycle times on the fetching and decoding logic. In many systems the high level architecture is unchanged from earlier scalar designs. The superscalar designs use instruction level parallelism for improved implementation of these architectures.

A good example of a superscalar processor is the IBM RS/6000 [10]. There are three major subsystems in this processor: the instruction fetch unit, an integer processor, and a floating point processor. The instruction fetch unit is a 2- stage pipeline; during the first stage a packet of four instructions is fetched from an instruction cache, and in the second

⁷In the Linpack benchmark tables the theoretical peak performance of the Cray-1S is listed as 160 MFLOPS, probably because it was realistic to keep only two pipelines chained together for any reasonable period of time

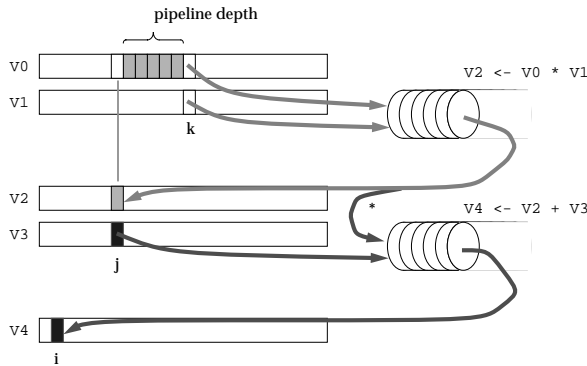


Figure 16: Vector Chaining

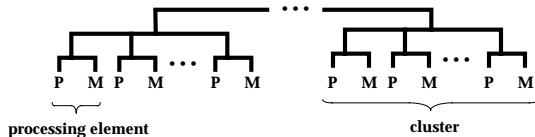


Figure 17: Mesh cm*

into vector instructions. A superscalar machine still requires a very sophisticated compiler to allocate resources and schedule operations in an order that will best take advantage of the resources of the machine, but in the long run the superscalar approach may be more flexible and applicable to a wider range of applications than vector processing.

3.5.3 Shared Memory MIMD Multiprocessors

The history of shared memory multiprocessors goes back to the early 1970s, to two influential research projects at Carnegie Mellon University. The first machine, named c.mmp (from the PMS notation for “computer with multiple mini-processors”), was organized around a crossbar switch that connected 16 PDP-11 processors to 16 memory banks. The second, cm*, also used PDP-11 processors, but connected them via the tree-shaped network shown in Figure 17 on page 63. The basic building block for this system was a processor cluster, which consisted of four processors, each with their own local memories. The global memory space was evenly partitioned among the memories in the system. When a processor generated a request for address i , its bus logic would check to see if i was in the range of addresses in that machine’s local memory. If it wasn’t, the request was transferred to a cluster controller, which would see if i belonged to any other memory within that cluster. If not, the request would be routed up the tree to another level of cluster controllers. In all, 50 processors were connected by three levels of buses.

cm* was an early example of a non-uniform memory access (NUMA) architecture. Depending on whether an item was in a processor’s local memory, within the same cluster, or in another cluster, the time to fetch an item was $3\mu s$, $9\mu s$, or $27\mu s$, respectively. As a reference point, a PDP-11 of this era, without the cluster interconnection logic, could fetch an item from main memory in about $2\mu s$.

One of the first commercial systems of this type was the BBN Butterfly. As its name implies, it consisted of a butterfly switch connecting up to 256 processors and memories. The processors were Motorola 68000 single-chip microprocessors. BBN added an extra path from the processors to memory by pairing up each processor with one of the memory modules,

stage instructions are routed to the integer processor and/or floating point processor. An interesting feature of this instruction unit is that it executes branch instructions itself so that in a tight loop there is effectively no overhead from branching since the instruction unit executes branches while the data units are computing values. The integer unit is a four-stage pipeline. In addition to executing data processing instructions this unit does some preprocessing for the floating point unit. The floating point unit itself is six stages deep.

The following example from [10] shows the potential of this style of computing. This code from a computer graphics application rotates and displaces a set of (x, y) pairs by an angle q and displacement (x_d, y_d) :

$$\begin{aligned} x'_i &= x_i \cos \Theta - y_i \sin \Theta + x_d \\ y'_i &= y_i \cos \Theta - x_i \sin \Theta + y_d \end{aligned}$$

A vector processor would load the (x, y) pairs into two vector registers and then use vector instructions. On the RS/6000 the operations are compiled into the following loop (constants x_d , $\sin \Theta$, etc. are loaded into registers before the loop begins):

```
L:    load  R8,x[i]
      fma  R10,R8,cos,xd
      load R9,y[i]
      fma  R11,R9,cos,yd
      fma  R12,R9,-sin,R10
      store R12,x[i]'
      fma  R13,R8,sin,R11
      store R13,y[i]'
      branch L
```

The **fma** W,X,Y,Z instruction is “floating multiply and add”, i.e. $W = X*Y+Z$. Note that the compiler has carefully interleaved load and store instructions with data processing instructions, and there are eight floating point operations (two per **fma** instruction) in each loop iteration and the loop itself has eight instructions, not counting the branch. Over the entire loop, then, the processor initiates one floating point operation per instruction. Since the instruction fetch unit executes the branch there are no cycles when the floating point unit is not busy. The machine will deliver one result per cycle for arbitrarily long vectors as long as there are no cache misses. See [10] for a detailed explanation of the timing of this loop. A 62.5 MHz RS/6000 system ran the LINPACK benchmark (Gaussian elimination) at a rate of 104 MFLOPS and has a theoretical peak performance of 125 MFLOPS. The HP 9000/735 workstation has a 99 MHz superscalar HP-PA processor. This machine executes the LINPACK benchmark at 107 MFLOPS, with a theoretical peak performance of 198 MFLOPS. By comparison, the Cray T-3E, with a clock cycle of 80MHz, performs at 110 MFLOPS and a theoretical peak of 160 MFLOPS. The advantage of the superscalar approach is that it does not rely on a vectorizing compiler to detect loops and turn them

so each processor had a “favored” memory unit. The processor could access this memory directly without going through the switch. The result was a NUMA architecture, with a ratio of about 15:1 in access times depending on whether the processor used the butterfly switch or the direct connection.

A recent commercial system in this category, with computing power and scalability that could potentially make it widely used in computational science, is the KSR-1 from Kendall Square Research. Processing elements are connected in rings, with from 8 to 32 PEs per ring. Larger systems have a second level ring that connects up to 34 first-level rings, for a maximum machine size of 1088 processors. Each ring is unidirectional, i.e. information flows in only one direction, with a bandwidth of 1 GB/sec.

Each PE has a 32MB cache, but there is no primary memory. This unusual organization uses a cache directory to access all information. When a processor makes a reference to an item in location i , the cache line that contains i migrates around the rings until it reaches the requesting processor. If two or more processors need an item the hardware implements the necessary cache coherency protocols to keep the items up to date. This type of system is also known as a *shared virtual memory*.

The processors in the KSR-1 are proprietary 64-bit superscalar processors with a 20MHz cycle time. According to the LINPACK benchmark report, a single KSR-1 processor achieves 31 MFLOPS out of a theoretical peak of 40 MFLOPS on 100×100 Gaussian elimination. A 32-node system reaches 513 MFLOPS, a speedup of a factor of 16.5.

3.5.4 Multiprocessor Vector Machines

Most of the vector supercomputer manufacturers produce multiprocessor systems based on their vector processors. Since a single node is so expensive and so finely tuned to memory bandwidth and other architectural parameters, the multiprocessor configurations have only a few processors. The largest currently is the Cray C-90, which has up to 16 processors.

An 8-processor Cray Y-MP is a shared memory MIMD system in the style of the BBN Butterfly, with processors connected to a set of memories via a multistage switching network. The switching network is a 3-level crossbar. A major difference between the switching networks in the Butterfly and Y-MP is that in the Y-MP there are many more memory modules than processors since the individual processors were designed to connect to an interleaved memory. There are enough memory modules — 32 per processor — and enough flexibility in the switch to allow each processor to connect to several banks at once so it can transfer vectors into and out of vector registers at a rate of one item per clock cycle. The assignment of virtual addresses to memory modules differs from the Butterfly arrangement, also, since in the interleaved organization consecutive addresses need to be in different modules. An exception to the rule that few processors are used in multiprocessor vector machines is a 222-processor system recently announced by Fujitsu. The nodes in this machine will be interconnected via a large, single-stage crossbar switch. Each node in the system consists of a local interleaved memory, a scalar processing unit, and a vector processing unit. The network interface implements a single address space from the individual local memories. Each node

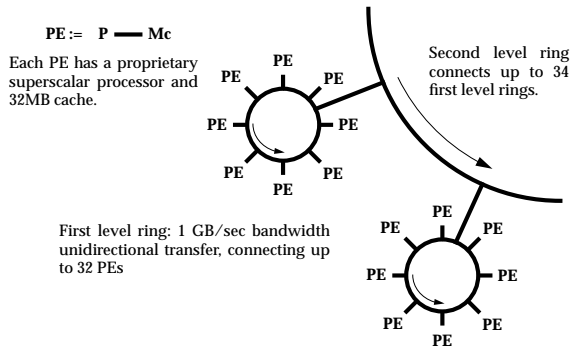


Figure 18: Kendall Square Research KSR-1

in the VPP500 will have a peak performance of about 1.5 GFLOPS, so a full 222 processor system will have a theoretical peak performance of over 300 GFLOPS.

3.5.5 Distributed Memory MIMD Systems

An early and very influential distributed memory parallel processor was the “Cosmic Cube”, a research project carried out by members of the Physics and Computer Science departments at Caltech [30]. This was one of the first systems to treat the interconnection network as a medium for exchanging messages, as opposed to an extended bus that simply fetched single words.

Each node in the Cosmic Cube was a single-board computer with an Intel 8086 processor chip, 8087 floating point coprocessor, and 128KB memory. 64 boards were interconnected as a 6-dimensional hypercube. Communication over the interconnection network was fairly slow, at 2Mbps per link, and used a store and forward protocol. Intel’s commercial version of the Cosmic Cube was the iPSC-1, which used 80286 processor chips, 512KB memory per node, and 10Mbps communication chips, and came in configurations from 16 to 128 processors (from 4D up to 7D hypercubes). Other commercial hypercubes of this era included the NCUBE-1 and FPS T-series.

The iPSC-2 was also a hypercube-based machine, but it incorporated “worm-hole routing” in place of the store and forward packet switching used in earlier systems. A worm-hole router uses a form of circuit switching to establish a communication path between two processors according to fixed rules. For example, in the two dimensional mesh the rule might be to use the vertical links first until the row of processors containing the destination processor is reached and to then use the horizontal links until the connection is made. Efficiency is improved because the technique removes the requirement that each processor along a route makes a decision about the direction of the next step of the communication. This in effect reduces the dependence of the diameter of the array on the number of steps required to transmit a data item from one end of the system to the other. What one gains in efficiency one loses in flexibility because worm-hole routing eliminates the opportunity to use alternate paths that might be provided by the network. For example, congestion on a single link may be unavoidable even though alternate paths are available to ease the congestion.

Following the iPSC/2 (and the iPSC/860, which was similar but used i860 RISC processors instead of 80386 processors at each node), Intel built a research machine known as the Touchstone Delta. A commercial system based on the Delta is the Paragon XP/S. The interconnection network is a 2D mesh instead of a hypercube, and uses specially designed message routing chips to improve communication bandwidth. Each node in the Paragon has two i860 processors, one for computation and the other for message handling. This second processor deals with incoming messages and other overhead so the main processor does not have to be interrupted to handle message traffic.

An interesting machine that is a hybrid with attributes of both SIMD and distributed memory MIMD machines is the CM-5 from Thinking Machines. The basic machine consists of a tree of processing nodes, where each node has a SPARC microprocessor, optional vector

processors, and up to 32MB of local memory. The interconnection network is based on the idea of a “fat tree,” a tree that has wider communication channels near the root in order to handle the higher volume of traffic expected to flow in that region of the network Figure 19 on page 67. Each communication link in the CM-5 has a bandwidth of 20 Mbps. There are two upward links from each leaf node. The links are attached to different switches, both for higher bandwidth and to provide alternative routes to avoid congestion in the network. First level interior switches have two upward links, but higher level switches have four upward links to implement the fat tree idea of higher bandwidth closer to the root.

The CM-5 has a control network consisting of a set of control processors interconnected with their another tree-shaped network. The control processors and their tree are a completely separate subsystem. Control processors are also SPARC microprocessors, but since they do little if any data processing they do not have as much memory or any vector coprocessors. It is the control network that allows the system to operate as an SIMD or SPMD machine by synchronizing sets of data processors when they are all working on the same program.

3.5.6 SIMD Machines

Several commercial SIMD machines were introduced in the 1970s, but they were not very widely used. Interest in this class of machines was renewed in the late 1980s with the introduction of the Connection Machine (CM-1) from Thinking Machines, Inc., and the MasPar MP-1. Part of the renewed interest is certainly the result of VLSI technology, which had advanced by that time to the point where several small processors could be put on a single chip. By themselves these processors were too simple to compete with general purpose single-chip processors such as the Motorola 68020 or Intel 80386, but literally thousands of them could be packaged in a small space and built into a cost-effective system. For example, 32 MP-1 processors fit on a single chip, and 32 chips were placed on a single board, for a total of 1024 processors (and their associated memory) in approximately 4 square feet.

The CM-1 was based on 1-bit processors. Every operation in the machine processed 1-bit operands and produced 1-bit results. Operations on larger data elements, for example 32-bit integers, required one cycle per bit. Attached to each processor was a local memory with a capacity of 4K bits. Memory references, like processor operations, were 1-bit operations, i.e. a fetch copied 1 bit from memory into a 1-bit processor register. 16 processors were implemented on a single chip. Within a chip, processors were connected with a grid, and up to 4096 chips were connected via a 12-dimensional hypercube. All processors obeyed instructions issued by a central control processor, which in turn was connected to a front-end workstation.

The MasPar MP-1 was introduced a few years after the CM-1. It also has a very narrow datapath, but it processes data 4 bits at a time instead of 1 bit at a time. Each processor can have up to 64KB of local memory. One of the interesting aspects of the MP-1 is that there are two separate communication systems, and programmers can alternate between them to choose the best performance for different parts of their algorithms. One interconnection

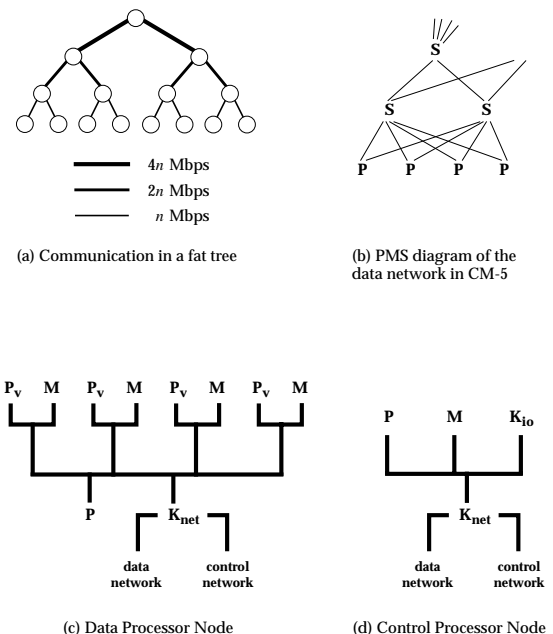


Figure 19: Thinking Machines Corporation CM-5

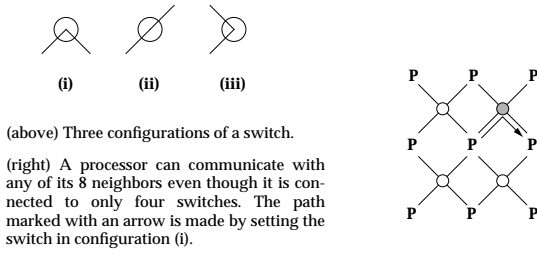


Figure 20: Switching in an X-net

network is known as the X-net (Figure 20). It connects each processor to its 8 nearest neighbors in a 2D mesh with wraparound connections. The other connection is a global router, which provides point-to-point communication between any two PEs. The router is implemented by a 3-stage switching network, where each stage in a 1024-processor machine contains a 16×16 crossbar; together the three stages comprise a 1024×1024 crossbar. The processors are controlled by a proprietary RISC processor known as the array control unit, or ACU. The ACU has its own local memory and is used for scalar operations, while the processor array is intended for vector and array operations. An MP-1 can be configured as an $n \times n$ square mesh or a $n \times 2n$ rectangular mesh. The smallest configuration has 1,024 processors and the largest has 16,384 processors in a 128×128 grid.

The newest machines from Thinking Machines and MasPar are the CM-5 and MP-2, respectively. The CM-5 is described in more detail in [15]. The MP-2 has a wider internal data path than the MP-1 — 32 bits vs. 4 bits — but is otherwise very similar to the MP-1 in that it uses both the X-net and global router to connect PEs in a 2D mesh. The largest MP-2, which has 16,384 (2^{14}) processors, has a theoretical peak performance of 550 MFLOPS and reaches 473 MFLOPS on the LINPACK benchmark for parallel machines.

3.6 Performance Models for Vector and Parallel Machines

Hockney and Jesshope [13] introduced two parameters to describe the performance of vector processors. The first parameter is the theoretical peak performance, or *asymptotic perfor-*

mance, denoted r_∞ . It is the maximum possible rate of computation, expressed as a number of floating point operations per second. The parameter may be applied to a single vector pipeline or to an entire system. Thus r_∞ for a single pipe of the Cray Y-MP is 167 MFLOPS (6ns cycle, one result per cycle), and approximately 2.6 GFLOPS for an 8-processor system with the add and multiply units in operation simultaneously. The other parameter, designated $n_{1/2}$ and known as the half performance length, is the length of the vector for which a system attains half of its peak performance, i.e. $0.5 \cdot r_\infty$. $n_{1/2}$ is a function of vector startup time and pipeline depth. As these values increase it becomes harder and harder to achieve near peak performance for the system because it requires algorithms with longer and longer vectors. As we saw in section 2.2, the startup times for the CDC vector computers were an order of magnitude greater than those from Cray Research. This is reflected in $n_{1/2}$ for the two systems differing also by an order of magnitude. Even though r_∞ was higher for the CDC machines, the systems from Cray proved more popular than those from CDC, so users seem to find lower $n_{1/2}$ more important.

Perhaps the most fundamental performance question that can be asked of an algorithm running on a parallel system is “does it run faster, and if so by how much?”. Ideally, if one uses P processors to solve a given problem, the execution time would be cut by a factor of P . This leads to a definition of *speedup*, which is the ratio of the execution time on one processor to the execution time on P processors:

$$S_P = \frac{\text{Execution time using one processor}}{\text{Execution time using } P \text{ processors}}$$

For example, if a program takes 18 minutes to run on one processor, but only 4.7 minutes on four processors, the speedup is a factor of $18.0/4.7 = 3.8$.

The strength of such a measure is that it uses observed execution time and thus takes into account any overhead in the parallel system for breaking a job into parallel tasks and intertask communication time. Comparing time on one processor vs. time on P processors can be misleading, however. One might be tempted to write a program for a P -processor machine, time it first on one processor and then on P processors, and call the ratio the speedup. Plotted for different values of P this procedure gives an accurate measure of the scalability of the algorithm used, but it does not answer the question how much faster a problem may be solved using P processors since a parallel algorithm usually incurs overheads that are not found in sequential algorithms. Ortega and Voigt [24] defined speedup as the ratio of the solution time for the best serial algorithm with that required by the parallel algorithm:

$$S_P = \frac{\text{Execution time for the best serial algorithm on one processor}}{\text{Execution time for parallel algorithm on } P \text{ processors}}$$

In the 1960's, Amdahl [1] noted that speedup is limited by the size of the portion of a problem that is not executed faster. For example, suppose a program that executes in 10.0 seconds contains a key subroutine that accounts for 80% of the execution time. The rest of the program uses 20% of the total time, or 2.0 seconds. If we use a more efficient version of

the subroutine that runs twice as fast, total execution time will drop to 6.0 seconds (8.0/2 seconds for the subroutine, 2.0 seconds for the remainder of the program). If we find a parallel subroutine that speeds up perfectly on P processors, and run the program on an 8-processor machine, execution time will drop to 3.0 seconds (8.0/8 seconds for the parallel portion, 2.0 seconds for the sequential portion). If we run on 100 processors, the total execution time will be 2.08 seconds. As more processors are used, the execution time gets closer to the time required for the sequential part, but it can never get lower than this. Since the fastest this program will ever run is 2.0 seconds, no matter how many processors are used, the maximum speedup is a factor of 5.0.

If we normalize the formula that defines speedup by letting the sequential execution time be 1 and expressing the other times as percentages of the sequential time, we derive the following formulation of Amdahl's law for parallel processors:

$$S = \frac{1}{(1-a) + a/S_P}$$

This version of the equation makes the contribution of the sequential portion of the computation more apparent. Here a is the fraction of the program that can be performed in parallel, and thus $(1-a)$ is the portion that is sequential. The denominator is the time to execute the program in parallel, i.e. the sum of the time spent in the sequential portion and the time spent in the parallel part, where the parallel time is a function of the speedup factor of the parallel portion. If the parallel portion exhibits perfect speedup, i.e. a factor of P when run on P processors, the equation becomes:

$$S = \frac{1}{(1-a) + a/P}$$

The efficiency of a parallel computation is the ratio of the speedup to the number of processors used to obtain that speedup:

$$E_P = \frac{S_P}{P}$$

For example, if 10 processors are used and the program runs 10 times faster, we are running at the maximum possible speed, i.e. all processors are being used to their full capacity. If the speedup is only a factor of 5, however, the efficiency is $5/10 = 0.5$, i.e. half the computing power is lost in overhead or synchronization.

The above models do not try to characterize the execution of a parallel program. They simply measure the time required to execute a program on a given machine, and compare that time to sequential execution times. A simple model that breaks a parallel program into constituent parts is

$$T = t_{comp} + t_{comm} + t_{sync}$$

The three components of overall execution time are t_{comp} , the computation time, t_{comm} , the time spent in communication, and t_{sync} , the time used to synchronize the processors at

appropriate points in the algorithm. This type of analysis is very important for algorithms that will run on distributed memory machines, where locality and communication costs will play a major role in efficiency.

Expressions for communication complexity can range from very simple, e.g. all communication requires the same amount of time (a reasonable model in some cases for a shared memory system), to very complex, as would be required for an accurate model of a distributed memory system that communicates by message passing. In the former case, the number of accesses to memory times the average access time might suffice. In the latter case, a more complex analysis is necessary for an accurate model. For example, the time to send a message from processor i to processor j in a packet switched network is often modeled by an expression such as

$$a + b \cdot n_p$$

where a is the overhead in setting up the message in the sending processor (and, if necessary, storing a message in the receiving processor), b is the distance from i to j , and n_p is the length of the message in packets. Again, this simple formula hides many additional complexities that might or might not effect performance: the existence of a separate processor at each processing node to handle messages, contention at links or nodes, whether or the routing is fixed or dynamic, etc.

The overhead for synchronization can also involve a number of issues depending on the algorithm and the type of synchronization mechanism used. For example, an SIMD system automatically synchronizes parallel subtasks and no further modeling is required. For MIMD systems, in addition to the overhead associated with the actual synchronization process, there is also the idle time created when processors wait at a synchronization point. A thorough discussion of synchronization mechanisms may be found in Andrews and Schneider [1983]. The development of a complexity model involving all three of the above factors in great detail may be found in Reed and Patrick [1985]. A general discussion of many of the issues relative to linear algebra algorithms may be found in Ortega [1988].

4 Exercises

Exercise 1 *The creation of a PMS diagram for one of the computer systems that you currently use.*

Draw a PMS diagram for one of the systems you use. The diagram should include at least the size of main memory and cache, the processor(s), and the pathways between processor(s) and memory. Consult [32] for more information on PMS if necessary.

Exercise 2 *The analysis of a computer system.*

Write a program that creates two $n \times n$ matrices of random real numbers between 0 and 1 and then computes the product of the two matrices. n should be an input parameter. The only output from the program should be the time required to multiply the matrices.

- Analyze the complexity of your program, i.e. how many operations does it perform as a function of the input size n ?
- Construct a table of execution times as a function of the matrix size n . Keep adding entries until you have runs that take more than a few minutes. Do the entries in the table agree with your predicted performance model?
- Plot the data in the table with `gnuplot` or one of the other plotting packages described in Chapter “Scientific Visualization In High Performance Computing”.
- From the data in the table can you tell when the matrices are large enough so they don't all fit in cache? How does this data point correlate with the size of the cache on your system?
- If you have a vector processor, can you infer anything about optimum vector sizes from the pattern of data in your table?
- If you are collecting your data on a workstation, record the system load average along with the execution time. Try several runs of the same matrix size but at times when the load average is high, medium, and low. Plot the data again, either with separate lines for different load averages, or with error bars that show the range of times. Does system load affect the performance of your program?

Exercise 3 *A comparison of your calculated values versus the manufacturer's reference values.*

Look up “MIPS” rating of your machine, either in the manuals provided by the manufacturer or in a standard reference ([find Dongarra's numbers]). How does it compare to numbers you achieved in previous problem?

Exercise 4 *Communication bandwidth between frame buffer and monitor.*

What is the communication bandwidth between the frame buffer and the monitor in a typical high resolution 8-bit RGB display?

Exercise 5 *Number of unique n -bit strings.*

Use mathematical induction to prove there are 2^n unique n -digit strings composed only of the symbols 1 and 0.

Exercise 6 *Representation of a negative integer.*

Prove that if $b = b_0b_1 \dots b_{n-1}$ is the n -bit representation of the integer x , the two's complement of b found by inverting every bit b_i and adding 1, is the representation of $-x$. Hint: the value of the complement of b_i is $1 - b_i$.

Exercise 7 *Bit shifts and mathematical operations.*

(a) Prove that if $b = b_0b_1 \dots b_{n-1}$ is the n -bit representation of the integer x , then (a) shifting b left by i bits is equivalent to multiplying x by 2^i and (b) shifting b right by i bits is equivalent to dividing x by 2^i (ignoring any remainder).

(b) If $b_0 = 1$, b represents a negative number in a two's complement system. In what is known as a “logical shift right” 0's are inserted into the leftmost bits, which means the result will be a positive number. Obviously if we divide a negative number by a power of two we expect a negative result, and in this case the logical shift doesn't give the correct answer. For example, $-16 \div 4 = -4$. The 8-bit two's complement representation of -16 is 11110000. If we shift it right by two bits to divide by 4, we get 00111100, which represents 60, not -4 . Can you think of a simple way to “fix” the shift operation so that it gives the correct result when it shifts both positive and negative numbers?

Exercise 8 *Low order bits and division by powers of 2.*

Prove that if $b = b_0b_1 \dots b_{n-1}$ is the n -bit representation of the integer x , the low order i bits are the value of the remainder of $b \div 2^i$. NOTE: this operation is also performed very efficiently on most machines. Let m be a pattern known as a “mask” that contains 0's in the high order $n-i$ bits and 1's in the low order i bits. An operation known as a “bitwise AND” will compute a pattern x such that $x_i = b_i \wedge m_i$ (the \wedge operation is the logical AND, which is 1 if and only if both operands are 1). To find the remainder of a division by 2^i , create a mask with i 1's in the low order bits, then “and” it with b . For example, the remainder of 14 (00001110 in an 8-bit system) divided by $4 = 2^2$ is 00001110 \wedge 00000011 = 00000010 = 2_{10} .

Exercise 9 *Floating point numbers on your system.*

Explain the floating point number system on the machine that you are using.

Look up the representation for floating point numbers in the system you will be using for this course. Does it conform to the IEEE standard (standard 754)? How many bits are in the mantissa and exponent in single precision? In double precision? Does a double precision number really have twice as much precision as a single precision number? Explain.

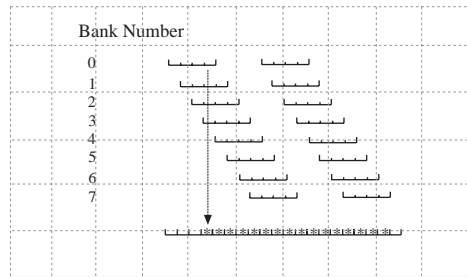


Figure 21: Gantt chart for 8-way interleaved memory

Exercise 10 *An analysis of memory cycle time.*

A Gantt chart can be used to show how interleaved memory works by drawing one row for each memory bank. Mark the time line in units of processor cycles. If a processor requests an item from bank b at time t , draw a line in row b starting at time t and continuing for n units, where n is the memory cycle time. Figure 21 shows the Gantt chart for an 8-way interleaved memory in a system where the processor cycle time is 10ns and the memory cycle time is 40ns. The chart illustrates which memories are busy when the processor requests items from successive memory cells. Asterisks on the time line indicate when data items reach the processor (assuming data is delivered on the last memory cycle). Asterisks in every column indicate the memory is performing at its full potential, i.e. there are no bank conflicts.

Use a Gantt chart to show that for a system with 16 memory banks, 12.5ns processor cycle time, and 50ns memory cycle time, there will be no conflicts when the stride is 2 or 4, but there will be conflicts when the stride is 8 or 16. (NOTE: these cycle times and interleaving factors are taken from the Cray-1).

Exercise 11 *An illustration of vector chaining.*

Use a Gantt chart to illustrate the effectiveness of vector chaining. Use the following parameters: the length of each vector is 64 elements; the multiply pipeline is 7 stages deep, the add pipeline has 6 stages, and there is a one-cycle delay (called the chain slot time) after producing the first product before the first pair of operands go to the adder. How many cycles does it take to compute $V4 = V3 + (V0 * V1)$ without chaining? With chaining?

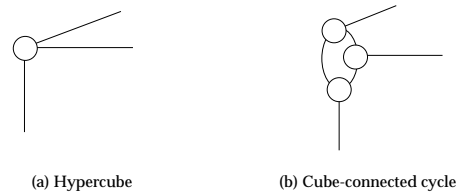


Figure 22: A single vertex in a 3D cube

Exercise 12 *Registers and the performance of the SAXPY benchmark.*

What is your understanding of the connection between scalar registers, vector registers, and floating point units for the performance of the SAXPY benchmark?

Does vector chaining improve performance on the SAXPY benchmark? What could you infer about the organization of the data path and the connection between scalar registers, vector registers, and floating point units if you measured the performance of SAXPY and found out that operands were indeed being chained between data units?

Exercise 13 *A cosideration of a parallel machine with a hypercube topology.*

Each vertex of a d -dimensional hypercube is connected to d other vertices. In a parallel machine with a hypercube topology, there is a single processor at each node, and it is linked to d other processors. An alternative design is to place a ring of d processors at each vertex, linking the i th processor to the neighboring vertex along dimension i . The result is a topology known as a *cube-connected cycle* (CCC) (Figure 22). Every node in a CCC is connected to 3 other nodes: its two neighbors in the ring plus the node in the neighboring vertex. Thus a CCC has a constant degree, no matter how many nodes are in the topology.

(a) What is the diameter of a CCC (assume bidirectional communication).

(b) How many nodes are in a general n -dimensional CCC?

(c) Draw a picture of a 4-dimensional CCC.

Exercise 14 *Same as above, but for Mayfly's hexagonal mesh.*

Exercise 15 *A determination of the paths for some messages for a 3D and 4D hypercube.*

Label the nodes in the 3d and 4d hypercubes in Figure 11. In the 4D cube, draw the paths that are taken by the following messages: from 0 to 15; from 3 to 12; from 5 to 10, and from 10 to 5.

Exercise 16 *A determination of the two different paths taken by messages between two processors in a hypercube.*

You may have noticed in the previous exercise that the message from node 5 to node 10 travels a different path than the message from node 10 to node 5. Explain why, for any two processors i and j , messages sent from i to j travel a different path than messages from j to i . Can you characterize the relationship between the two paths?

Exercise 17 *A determination of the processor-memory connections for a butterfly switch arrangement.*

For each interior node in the butterfly switch of figure Figure 15, indicate whether it is in the straight- through or flipped configuration for the following pairs of processor-memory connections. Assume the connections that come first in the list are made first by the switching network:

P0-M3, P1-M5, P2-M2, P3-M7, P4-M6, P5-M0, P6-M4, P7-M1

How many of these connections block due to contention in the switch?

Exercise 18 *A determination of the PMS diagram for the crossbar switch connecting the processing elements of a MasPar MP-1 and a Fujitsu VPP500.*

Most of the uses of a crossbar switch mentioned in this chapter connect a set of processors on the “input” side to a set of memories on the “output” side (Figure 14). However, two systems, the Fujitsu VPP500 and the MasPar MP-1, use a crossbar to interconnect processing elements (PEs), nodes which consist of a processor and its local memory. Draw a PMS diagram of such a system and explain how information could be moved from one node to another.

Exercise 19 *A consideration of some of the communication that takes place on a KSR-1.*

The ring network in the KSR-1 is a slotted ring, which means packets flow around the ring in discrete steps under control of a global clock. The KSR-1 transfers 128 million packets per second past each slot on the ring.

- The bandwidth is 1GB/sec. How wide is the communication channel?
- The processor cycle time is 20MHz. How many packets can be taken off the ring during each processor cycle?
- What is the diameter of a full-size (1088-PE) KSR-1?
- Ignoring communication overhead and the possibility of contention that would delay a message, how long will it take the machine to transfer 256 bytes along the longest communication path?

Exercise 20 *A plot of Amdahl's law.*

Use gnuplot or some other plotting package to plot Amdahl's law for different values of P . Note that α varies from 0 to 1, i.e. from completely sequential to completely parallel.

References

- [1] Amdahl, G., *The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*, AFIPS Conf. Proc. 30, pp. 483–485, 1967.
- [2] Andrews, G., and Schneider, F., *Concepts and Notations for Concurrent Programming*, Computing Surveys, Vol. 15, pp. 3–43, 1983.
- [3] Bell, G., *The Future of High Performance Computers in Science and Engineering*, Comm. ACM, Vol. 32, pp. 1091–1101, 1989.
- [4] Bhuyan, L., Yang, Q., and Agrawal, D., *Performance of Multiprocessor Interconnection Networks*, Computer, Vol. 22, No. 2, pp. 25–37, 1989.
- [5] Bouknight, W.J., et al., *The ILLIAC-IV System*, Proc. IEEE, April 1972, pp. 369–388. (reprinted in CSPE)
- [6] Buzbee, B., Remarks for the IFIP Congress '83 Panel on How to Obtain High Performance for High Speed Processors, Los Alamos National Laboratory Report LA-UR-84-1392, Los Alamos, NM, 1983.
- [7] Denning, P. and Tichy, W., *Highly Parallel Computation*, RIACS Report TR-90.35, NASA Ames Research Center, Moffet Field, CA, August, 1990.
- [8] Dongarra, J.J., *Performance of Various Computers Using Standard Linear Equations Software*, Tech Report CS-89-85, Univ. of Tennessee. [A continually updated report listing the performance of several hundred machines (from Cray C90 to Atari ST) on LINPACK benchmarks.]
- [9] Flynn, M., *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput., Vol. C-21, pp. 94, 1972.
- [10] Grohoski, G.F., *Machine Organization of the IBM RISC System/6000 Processor*, IBM J. Res. Develop. 43(1), January 1990, pp. 37–58. [Detailed explanation of superscalar execution in a modern RISC processor.]
- [11] Hennessy, J.L., and Patterson, D. A., *Computer Architecture: A Quantitative Approach*, Morgan-Kaufman, 1990. [Excellent modern text on basic computer architecture; rapidly becoming the standard text in senior undergraduate level computer science courses.]
- [12] Hennessy, J.L., *VLSI Processor Architecture*, IEEE Trans. Comp. C-33(12), December 1984, pp. 1221–1246. [Excellent article on the interaction between VLSI technology and computer processor design. In-depth discussion of then-emerging RISC designs and alternatives. 8–960.]

- [13] Hockney, R., and Jesshope, C., *Parallel Computers 2*, Adam Hilger, Ltd., Bristol, United Kingdom, 1988. [A good resource for computational scientists, with a nice history of high performance computing and comprehensive survey of parallel algorithms for important matrix operations in addition to parallel and vector computer architecture.]
- [14] Hwang, K., *Advanced Parallel Processing with Supercomputer Architectures*, Proc. IEEE, Vol. 75, pp. 1348–1379, 1987.
- [15] Hwang, K., *Advanced Computer Architecture*, McGraw-Hill, 1993. [A very recent book on parallel processing and high performance computing; a good reference for facts about recent machines, including CM-5, KSR-1, and Paragon X/PS.]
- [16] Gannon, D., and Van Rosendale, J., *On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms*, IEEE Trans. Comput., Vol. C-33, pp. 1180–1194, 1984.
- [17] Karp, A., *Programming for Parallelism*, Computer, Vol. 20, No. 5, pp. 43–57, 1987.
- [18] Kogge, P., *The Architecture of Pipelined Computers*, McGraw-Hill, 1981. [Dated, but excellent in-depth coverage of pipelined processors.]
- [19] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures*, Morgan-Kaufman, 1982. [A comprehensive theoretical view of architectures and algorithms; a good reference for parallel algorithms and interconnection networks.]
- [20] Metcalfe, R.M., and Boggs, D.R., *Ethernet: Distributed Packet Switching for Local Computer Networks*, Comm. ACM 19(7), July 1976, pp. 395–404. [reprinted in CSPE]
- [21] Minsky, M., *Form and Content in Computer Science*, J. ACM, Vol. 17, pp. 197–215, 1970.
- [22] Nitzberg, B., and Lo, V., *Distributed Shared Memory: A Survey of Issues and Algorithms*, Computer, Vol. 24, No. 8, pp. 52–60, 1991.
- [23] Ortega, J., *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, NY, 1988.
- [24] Ortega, J., and Voigt, R., *Solution of Partial Differential Equations on Vector and Parallel Computers*, SIAM, Philadelphia, PA, 1985.
- [25] Patterson, D.A., *Reduced Instruction Set Computers*, Comm. ACM 28(1), January 1985, pp. 8–20. [Discussion of RISC principles, with good explanation of instruction pipelines and how RISC can take advantage of them.]

- [26] Reed, D. and Grunwald, D., *The Performance of Multicomputer Interconnection networks*, Computer, Vol. 20, No. 26, pp. 63–73, 1987.
- [27] Reed, D. and Patrick, M., *Parallel Iterative Solution of Sparse Linear Systems: Models and Architectures*, Parallel Computing, Vol. 2, No. 1, pp. 45–68, 1985.
- [28] Russel, R.M., *The Cray-1 Computer System*, Comm. ACM 21(1), January 1978, pp. 63–72. [reprinted in CSPE]
- [29] Schwartz, J., *Ultracomputers*, ACM Trans. Prog. Lang. Syst., Vol. 2, pp. 484–521.C, 1980.
- [30] Seitz, L., *The Cosmic Cube*, Comm. ACM 28(1), January 1985, pp. 22–33. [Overview of research prototype that later became the Intel iPSC-1.]
- [31] Siegel, H., *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, MA, 1985.
- [32] Siewiorek, D.P., Bell, C.G., and Newell, A., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982. [A collection of original chapters and primary source material on historic architectures and networks, including IBM 360, Cray-1, ILLIAC-IV, c.mmp and cm*, PDP-11, Intel 8086, Alohanet, and Ethernet.]
- [33] Stone, H.S., *High Performance Computer Architecture*, Addison-Wesley, Reading, MA, 1993 (3rd ed.) [A good book for computational scientists; in addition to detailed explanations of pipelining and memory organization (suitable for graduate level courses in computer science) there are chapters on scientific applications, vector machines, and parallel processing.]
- [34] Trew, A. and Wilson, A., Eds., *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*, Springer-Verlag, New York, NY, 1991.
- [35] Ware, W., *The Ultimate Computer*, IEEE Spectrum, Vol. 10, No. 3, pp. 89–91, 1973.