High Performance Computing

By:
Charles Severance

High Performance Computing

By:
Charles Severance

Online:
< http://cnx.org/content/col11136/1.2/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: November 13, 2009
PDF generated: November 13, 2009
For copyright and attribution information for the modules contained in this collection, see p. 118.

Table of Contents

Chapter 1

What is High Performance Computing?

1.1 Introduction to the Connexions Edition[1]

1.1.1 Introduction to the Connexions Edition

The purpose of this book has always been to teach new programmers and scientists about the basics of High Performance Computing. Too many parallel and high performance computing books focus on the architecture, theory and computer science surrounding HPC. I wanted this book to speak to the practicing Chemistry student, Physicist, or Biologist who need to write and run their programs as part of their research. I was using the first edition of the book written by Kevin Dowd in 1996 when I found out that the book was going out of print. I immediately sent an angry letter to O'Reilly customer support imploring them to keep the book going as it was the only book of its kind in the marketplace. That complaint letter triggered several conversations which let to me becoming the author of the second edition. In true "open-source" fashion - since I complained about it - I got to fix it. During Fall 1997, while I was using the book to teach my HPC course, I re-wrote the book one chapter at a time, fueled by multiple late-night lattes and the fear of not having anything ready for the weeks lecture.

The second edition came out in July 1998, and was pretty well received. I got many good comments from teachers and scientists who felt that the book did a good job of teaching the practitioner - which made me very happy.

In 1998, this book was published at a crossroads in the history of High Performance Computing. In the late 1990's there was still a question a to whether the large vector supercomputers with their specialized memory systems could resist the assault from the increasing clock rates of the microprocessors. Also in the later 1990's there was a question whether the fast, expensive, and power-hungry RISC architectures would win over the commodity Intel microprocessors and commodity memory technologies.

By 2003, the market had decided that the commodity microprocessor was king - its performance and the performance of commodity memory subsystems kept increasing so rapidly. By 2006, the Intel architecture had eliminated all the RISC architecture processors by greatly increasing clock rate and truly winning the increasingly important Floating Point Operations per Watt competition. Once users figured out how to effectively use loosely coupled processors, overall cost and improving energy consumption of commodity microprocessors became overriding factors in the market place.

These changes led to the book becoming less and less relevant to the common use cases in the HPC field and led to the book going out of print - much to the chagrin of its small but devoted fan base. I was reduced to buying used copies of the book from Amazon in order to have a few copies laying around the office to give as gifts to unsuspecting visitors.

Thanks the the forward-looking approach of O'Reilly and Associates to use Founder's Copyright and releasing out-of-print books under Creative Commons Attribution, this book once again rises from the

[1] This content is available online at <http://cnx.org/content/m32709/1.1/>.

ashes like the proverbial Phoenix. By bringing this book to Connexions and publishing it under a Creative Commons Attribution license we are insuring that the book is never again obsolete. We can take the core elements of the book which are still relevant and a new community of authors can add to and adapt the book as needed over time.

Publishing through Connexions also keeps the cost of printed books very low and so it will be a wise choice as a textbook for college courses in High Performance Computing. The Creative Commons Licensing and the ability to print locally can make this book available in any country and any school in the world. Like Wikipedia, those of us who use the book can become the volunteers who will help improve the book and become co-authors of the book.

I need to thank Kevin Dowd who wrote the first edition and graciously let me alter it from cover to cover in the second edition. Mike Loukides of O'Reilly was the editor of both the first and second editions and we talk from time to time about a possible future edition of the book. Mike was also instrumental in helping to release the book from O'Reilly under Creative Commons Attribution. The team at Connexions has been wonderful to work with. We share a passion for High Performance Computing and new forms of publishing so that the knowledge reaches as many people as possible. I want to thank Jan Odegard and Kathi Fletcher for encouraging, supporting and helping me through the re-publishing process. Daniel Williamson did an amazing job of converting the materials from the O'Reilly formats to the Connexions formats.

I truly look forward to seeing how far this book will go now that we can have an unlimited number of co-authors to invest and then use the book. I look forward to work with you all.

Charles Severance - November 12, 2009

1.2 Introduction to High Performance Computing[2]

1.2.1 What Is High Performance Computing

1.2.1.1 Why Worry About Performance?

Over the last decade, the definition of what is called high performance computing has changed dramatically. In 1988, an article appeared in the Wall Street Journal titled "Attack of the Killer Micros" that described how computing systems made up of many small inexpensive processors would soon make large supercomputers obsolete. At that time, a "personal computer" costing $3000 could perform 0.25 million floating-point operations per second, a "workstation" costing $20,000 could perform 3 million floating-point operations, and a supercomputer costing $3 million could perform 100 million floating-point operations per second. Therefore, why couldn't we simply connect 400 personal computers together to achieve the same performance of a supercomputer for $1.2 million?

This vision has come true in some ways, but not in the way the original proponents of the "killer micro" theory envisioned. Instead, the microprocessor performance has relentlessly gained on the supercomputer performance. This has occurred for two reasons. First, there was much more technology "headroom" for improving performance in the personal computer area, whereas the supercomputers of the late 1980s were pushing the performance envelope. Also, once the supercomputer companies broke through some technical barrier, the microprocessor companies could quickly adopt the successful elements of the supercomputer designs a few short years later. The second and perhaps more important factor was the emergence of a thriving personal and business computer market with ever-increasing performance demands. Computer usage such as 3D graphics, graphical user interfaces, multimedia, and games were the driving factors in this market. With such a large market, available research dollars poured into developing inexpensive high performance processors for the home market. The result of this trend toward faster smaller computers is directly evident as former supercomputer manufacturers are being purchased by workstation companies (Silicon Graphics purchased Cray, and Hewlett-Packard purchased Convex in 1996).

As a result nearly every person with computer access has some "high performance" processing. As the peak speeds of these new personal computers increase, these computers encounter all the performance

[2]This content is available online at <http://cnx.org/content/m32676/1.1/>.

challenges typically found on supercomputers.

While not all users of personal workstations need to know the intimate details of high performance computing, those who program these systems for maximum performance will benefit from an understanding of the strengths and weaknesses of these newest high performance systems.

1.2.1.2 Scope of High Performance Computing

High performance computing runs a broad range of systems, from our desktop computers through large parallel processing systems. Because most high performance systems are based on *reduced instruction set computer* (RISC) processors, many techniques learned on one type of system transfer to the other systems.

High performance RISC processors are designed to be easily inserted into a multiple-processor system with 2 to 64 CPUs accessing a single memory using *symmetric multi processing* (SMP). Programming multiple processors to solve a single problem adds its own set of additional challenges for the programmer. The programmer must be aware of how multiple processors operate together, and how work can be efficiently divided among those processors.

Even though each processor is very powerful, and small numbers of processors can be put into a single enclosure, often there will be applications that are so large they need to span multiple enclosures. In order to cooperate to solve the larger application, these enclosures are linked with a high-speed network to function as a *network of workstations* (NOW). A NOW can be used individually through a batch queuing system or can be used as a large multicomputer using a message passing tool such as *parallel virtual machine* (PVM) or *message-passing interface* (MPI).

For the largest problems with more data interactions and those users with compute budgets in the millions of dollars, there is still the top end of the high performance computing spectrum, the scalable parallel processing systems with hundreds to thousands of processors. These systems come in two flavors. One type is programmed using message passing. Instead of using a standard local area network, these systems are connected using a proprietary, scalable, high-bandwidth, low-latency interconnect (how is that for marketing speak?). Because of the high performance interconnect, these systems can scale to the thousands of processors while keeping the time spent (wasted) performing overhead communications to a minimum.

The second type of large parallel processing system is the *scalable non-uniform memory access* (NUMA) systems. These systems also use a high performance inter-connect to connect the processors, but instead of exchanging messages, these systems use the interconnect to implement a distributed shared memory that can be accessed from any processor using a load/store paradigm. This is similar to programming SMP systems except that some areas of memory have slower access than others.

1.2.1.3 Studying High Performance Computing

The study of high performance computing is an excellent chance to revisit computer architecture. Once we set out on the quest to wring the last bit of performance from our computer systems, we become more motivated to fully understand the aspects of computer architecture that have a direct impact on the system's performance.

Throughout all of computer history, salespeople have told us that their compiler will solve all of our problems, and that the compiler writers can get the absolute best performance from their hardware. This claim has never been, and probably never will be, completely true. The ability of the compiler to deliver the peak performance available in the hardware improves with each succeeding generation of hardware and software. However, as we move up the hierarchy of high performance computing architectures we can depend on the compiler less and less, and programmers must take responsibility for the performance of their code.

In the single processor and SMP systems with few CPUs, one of our goals as programmers should be to stay out of the way of the compiler. Often constructs used to improve performance on a particular architecture limit our ability to achieve performance on another architecture. Further, these "brilliant" (read obtuse) hand optimizations often confuse a compiler, limiting its ability to automatically transform our code to take advantage of the particular strengths of the computer architecture.

As programmers, it is important to know how the compiler works so we can know when to help it out and when to leave it alone. We also must be aware that as compilers improve (never as much as salespeople claim) it's best to leave more and more to the compiler.

As we move up the hierarchy of high performance computers, we need to learn new techniques to map our programs onto these architectures, including language extensions, library calls, and compiler directives. As we use these features, our programs become less portable. Also, using these higher-level constructs, we must not make modifications that result in poor performance on the individual RISC microprocessors that often make up the parallel processing system.

1.2.1.4 Measuring Performance

When a computer is being purchased for computationally intensive applications, it is important to determine how well the system will actually perform this function. One way to choose among a set of competing systems is to have each vendor loan you a system for a period of time to test your applications. At the end of the evaluation period, you could send back the systems that did not make the grade and pay for your favorite system. Unfortunately, most vendors won't lend you a system for such an extended period of time unless there is some assurance you will eventually purchase the system.

More often we evaluate the system's potential performance using *benchmarks*. There are industry benchmarks and your own locally developed benchmarks. Both types of benchmarks require some careful thought and planning for them to be an effective tool in determining the best system for your application.

1.2.1.5 The Next Step

Quite aside from economics, computer performance is a fascinating and challenging subject. Computer architecture is interesting in its own right and a topic that any computer professional should be comfortable with. Getting the last bit of per- formance out of an important application can be a stimulating exercise, in addition to an economic necessity. There are probably a few people who simply enjoy matching wits with a clever computer architecture.

What do you need to get into the game?

- A basic understanding of modern computer architecture. You don't need an advanced degree in computer engineering, but you do need to understand the basic terminology.
- A basic understanding of benchmarking, or performance measurement, so you can quantify your own successes and failures and use that information to improve the performance of your application.

This book is intended to be an easily understood introduction and overview of high performance computing. It is an interesting field, and one that will become more important as we make even greater demands on our most common personal computers. In the high performance computer field, there is always a tradeoff between the single CPU performance and the performance of a multiple processor system. Multiple processor systems are generally more expensive and difficult to program (unless you have this book).

Some people claim we eventually will have single CPUs so fast we won't need to understand any type of advanced architectures that require some skill to program.

So far in this field of computing, even as performance of a single inexpensive microprocessor has increased over a thousandfold, there seems to be no less interest in lashing a thousand of these processors together to get a millionfold increase in power. The cheaper the building blocks of high performance computing become, the greater the benefit for using many processors. If at some point in the future, we have a single processor that is faster than any of the 512-processor scalable systems of today, think how much we could do when we connect 512 of those new processors together in a single system.

That's what this book is all about. If you're interested, read on.

Chapter 2

Memory

2.1 Introduction[1]

2.1.1 Memory

Let's say that you are fast asleep some night and begin dreaming. In your dream, you have a time machine and a few 500-MHz four-way superscalar processors. You turn the time machine back to 1981. Once you arrive back in time, you go out and purchase an IBM PC with an Intel 8088 microprocessor running at 4.77 MHz. For much of the rest of the night, you toss and turn as you try to adapt the 500-MHz processor to the Intel 8088 socket using a soldering iron and Swiss Army knife. Just before you wake up, the new computer finally works, and you turn it on to run the Linpack[2] benchmark and issue a press release. Would you expect this to turn out to be a dream or a nightmare? Chances are good that it would turn out to be a nightmare, just like the previous night where you went back to the Middle Ages and put a jet engine on a horse. (You have got to stop eating double pepperoni pizzas so late at night.)

Even if you can speed up the computational aspects of a processor infinitely fast, you still must load and store the data and instructions to and from a memory. Today's processors continue to creep ever closer to infinitely fast processing. Memory performance is increasing at a much slower rate (it will take longer for memory to become infinitely fast). Many of the interesting problems in high performance computing use a large amount of memory. As computers are getting faster, the size of problems they tend to operate on also goes up. The trouble is that when you want to solve these problems at high speeds, you need a memory system that is large, yet at the same time fast—a big challenge. Possible approaches include the following:

- Every memory system component can be made individually fast enough to respond to every memory access request.
- Slow memory can be accessed in a round-robin fashion (hopefully) to give the effect of a faster memory system.
- The memory system design can be made "wide" so that each transfer contains many bytes of information.
- The system can be divided into faster and slower portions and arranged so that the fast portion is used more often than the slow one.

Again, economics are the dominant force in the computer business. A cheap, statistically optimized memory system will be a better seller than a prohibitively expensive, blazingly fast one, so the first choice is not much of a choice at all. But these choices, used in combination, can attain a good fraction of the performance you would get if every component were fast. Chances are very good that your high performance workstation incorporates several or all of them.

[1] This content is available online at <http://cnx.org/content/m32733/1.1/>.
[2] See Chapter 15, Using Published Benchmarks, for details on the Linpack benchmark.

Once the memory system has been decided upon, there are things we can do in software to see that it is used efficiently. A compiler that has some knowledge of the way memory is arranged and the details of the caches can optimize their use to some extent. The other place for optimizations is in user applications, as we'll see later in the book. A good pattern of memory access will work with, rather than against, the components of the system.

In this chapter we discuss how the pieces of a memory system work. We look at how patterns of data and instruction access factor into your overall runtime, especially as CPU speeds increase. We also talk a bit about the performance implications of running in a virtual memory environment.

2.2 Memory Technology[3]

2.2.1 Memory Technology

Almost all fast memories used today are semiconductor-based.[4] They come in two flavors: *dynamic random access memory* (DRAM) and *static random access memory* (SRAM). The term *random* means that you can address memory locations in any order. This is to distinguish random access from serial memories, where you have to step through all intervening locations to get to the particular one you are interested in. An example of a storage medium that is *not* random is magnetic tape. The terms dynamic and static have to do with the technology used in the design of the memory cells. DRAMs are charge-based devices, where each bit is represented by an electrical charge stored in a very small capacitor. The charge can leak away in a short amount of time, so the system has to be continually refreshed to prevent data from being lost. The act of reading a bit in DRAM also discharges the bit, requiring that it be refreshed. It's not possible to read the memory bit in the DRAM while it's being refreshed.

SRAM is based on gates, and each bit is stored in four to six connected transistors. SRAM memories retain their data as long as they have power, without the need for any form of data refresh.

DRAM offers the best price/performance, as well as highest density of memory cells per chip. This means lower cost, less board space, less power, and less heat. On the other hand, some applications such as cache and video memory require higher speed, to which SRAM is better suited. Currently, you can choose between SRAM and DRAM at slower speeds — down to about 50 nanoseconds (ns). SRAM has access times down to about 7 ns at higher cost, heat, power, and board space.

In addition to the basic technology to store a single bit of data, memory performance is limited by the practical considerations of the on-chip wiring layout and the external pins on the chip that communicate the address and data information between the memory and the processor.

2.2.1.1 Access Time

The amount of time it takes to read or write a memory location is called the *memory access time*. A related quantity is the *memory cycle time*. Whereas the access time says how quickly you can reference a memory location, cycle time describes how often you can repeat references. They sound like the same thing, but they're not. For instance, if you ask for data from DRAM chips with a 50-ns access time, it may be 100 ns before you can ask for more data from the same chips. This is because the chips must internally recover from the previous access. Also, when you are retrieving data sequentially from DRAM chips, some technologies have improved performance. On these chips, data immediately following the previously accessed data may be accessed as quickly as 10 ns.

Access and cycle times for commodity DRAMs are shorter than they were just a few years ago, meaning that it is possible to build faster memory systems. But CPU clock speeds have increased too. The home computer market makes a good study. In the early 1980s, the access time of commodity DRAM (200 ns) was shorter than the clock cycle (4.77 MHz = 210 ns) of the IBM PC XT. This meant that DRAM could

[3] This content is available online at <http://cnx.org/content/m32716/1.1/>.

[4] Magnetic core memory is still used in applications where radiation "hardness" — resistance to changes caused by ionizing radiation — is important.

be connected directly to the CPU without worrying about over running the memory system. Faster XT and AT models were introduced in the mid-1980s with CPUs that clocked more quickly than the access times of available commodity memory. Faster memory was available for a price, but vendors punted by selling computers with *wait states* added to the memory access cycle. Wait states are artificial delays that slow down references so that memory appears to match the speed of a faster CPU — at a penalty. However, the technique of adding wait states begins to significantly impact performance around 25?33MHz. Today, CPU speeds are even farther ahead of DRAM speeds.

The clock time for commodity home computers has gone from 210 ns for the XT to around 3 ns for a 300-MHz Pentium-II, but the access time for commodity DRAM has decreased disproportionately less — from 200 ns to around 50 ns. Processor performance doubles every 18 months, while memory performance doubles roughly every seven years.

The CPU/memory speed gap is even larger in workstations. Some models clock at intervals as short as 1.6 ns. How do vendors make up the difference between CPU speeds and memory speeds? The memory in the Cray-1 supercomputer used SRAM that was capable of keeping up with the 12.5-ns clock cycle. Using SRAM for its main memory system was one of the reasons that most Cray systems needed liquid cooling.

Unfortunately, it's not practical for a moderately priced system to rely exclusively on SRAM for storage. It's also not practical to manufacture inexpensive systems with enough storage using exclusively SRAM.

The solution is a hierarchy of memories using processor registers, one to three levels of SRAM cache, DRAM main memory, and virtual memory stored on media such as disk. At each point in the memory hierarchy, tricks are employed to make the best use of the available technology. For the remainder of this chapter, we will examine the memory hierarchy and its impact on performance.

In a sense, with today's high performance microprocessor performing computations so quickly, the task of the high performance programmer becomes the careful management of the memory hierarchy. In some sense it's a useful intellectual exercise to view the simple computations such as addition and multiplication as "infinitely fast" in order to get the programmer to focus on the impact of memory operations on the overall performance of the program.

2.3 Registers[5]

2.3.1 Registers

At least the top layer of the memory hierarchy, the CPU registers, operate as fast as the rest of the processor. The goal is to keep operands in the registers as much as possible. This is especially important for intermediate values used in a long computation such as:

```
X = G * 2.41 + A / W - W * M
```

While computing the value of A divided by W, we must store the result of multiplying G by 2.41. It would be a shame to have to store this intermediate result in memory and then reload it a few instructions later. On any modern processor with moderate optimization, the intermediate result is stored in a register. Also, the value W is used in two computations, and so it can be loaded once and used twice to eliminate a "wasted" load.

Compilers have been very good at detecting these types of optimizations and efficiently making use of the available registers since the 1970s. Adding more registers to the processor has some performance benefit. It's not practical to add enough registers to the processor to store the entire problem data. So we must still use the slower memory technology.

[5]This content is available online at <http://cnx.org/content/m32681/1.1/>.

2.4 Caches[6]

2.4.1 Caches

Once we go beyond the registers in the memory hierarchy, we encounter caches. Caches are small amounts of SRAM that store a subset of the contents of the memory. The hope is that the cache will have the right subset of main memory at the right time.

The actual cache architecture has had to change as the cycle time of the processors has improved. The processors are so fast that off-chip SRAM chips are not even fast enough. This has lead to a multilevel cache approach with one, or even two, levels of cache implemented as part of the processor. Table 2.1 shows the approximate speed of accessing the memory hierarchy on a 500-MHz DEC 21164 Alpha.

Registers	2 ns
L1 On-Chip	4 ns
L2 On-Chip	5 ns
L3 Off-Chip	30 ns
Memory	220 ns

Table 2.1: Table 3-1: Memory Access Speed on a DEC 21164 Alpha

When every reference can be found in a cache, you say that you have a 100% hit rate. Generally, a hit rate of 90% or better is considered good for a level-one (L1) cache. In level-two (L2) cache, a hit rate of above 50% is considered acceptable. Below that, application performance can drop off steeply.

One can characterize the average read performance of the memory hierarchy by examining the probability that a particular load will be satisfied at a particular level of the hierarchy. For example, assume a memory architecture with an L1 cache speed of 10 ns, L2 speed of 30 ns, and memory speed of 300 ns. If a memory reference were satisfied from L1 cache 75% of the time, L2 cache 20% of the time, and main memory 5% of the time, the average memory performance would be:

```
(0.75 * 10 ) + ( 0.20 * 30 ) + ( 0.05 * 300 ) = 28.5 ns
```

You can easily see why it's important to have an L1 cache hit rate of 90% or higher.

Given that a cache holds only a subset of the main memory at any time, it's important to keep an index of which areas of the main memory are currently stored in the cache. To reduce the amount of space that must be dedicated to tracking which memory areas are in cache, the cache is divided into a number of equal sized slots known as *lines*. Each line contains some number of sequential main memory locations, generally four to sixteen integers or real numbers. Whereas the data within a line comes from the same part of memory, other lines can contain data that is far separated within your program, or perhaps data from somebody else's program, as in Figure 2.1 (Figure 3-1: Cache lines can come from different parts of memory). When you ask for something from memory, the computer checks to see if the data is available within one of these cache lines. If it is, the data is returned with a minimal delay. If it's not, your program may be delayed while a new line is fetched from main memory. Of course, if a new line is brought in, another has to be thrown out. If you're lucky, it won't be the one containing the data you are just about to need.

[6]This content is available online at <http://cnx.org/content/m32725/1.1/>.

Figure 3-1: Cache lines can come from different parts of memory

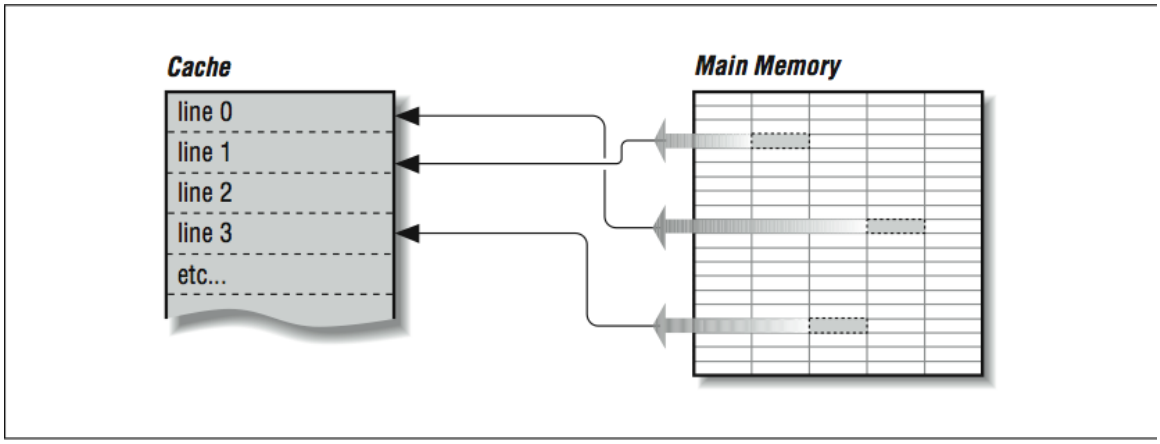


Figure 2.1

On multiprocessors (computers with several CPUs), written data must be returned to main memory so the rest of the processors can see it, or all other processors must be made aware of local cache activity. Perhaps they need to be told to invalidate old lines containing the previous value of the written variable so that they don't accidentally use stale data. This is known as maintaining *coherency* between the different caches. The problem can become very complex in a multiprocessor system.[7]

Caches are effective because programs often exhibit characteristics that help kep the hit rate high. These characteristics are called *spatial* and *temporal locality of reference*; programs often make use of instructions and data that are near to other instructions and data, both in space and time. When a cache line is retrieved from main memory, it contains not only the information that caused the cache miss, but also some neighboring information. Chances are good that the next time your program needs data, it will be in the cache line just fetched or another one recently fetched.

Caches work best when a program is reading sequentially through the memory. Assume a program is reading 32-bit integers with a cache line size of 256 bits. When the program references the first word in the cache line, it waits while the cache line is loaded from main memory. Then the next seven references to memory are satisfied quickly from the cache. This is called *unit stride* because the address of each successive data element is incremented by one and all the data retrieved into the cache is used. The following loop is a unit-stride loop:

```
DO I=1,1000000
  SUM = SUM + A(I)
END DO
```

When a program accesses a large data structure using "non-unit stride," performance suffers because data is loaded into cache that is not used. For example:

[7] Chapter 10, Shared-Memory Multiprocessors, describes cache coherency in more detail.

```
DO I=1,1000000, 8
  SUM = SUM + A(I)
END DO
```

This code would experience the same number of cache misses as the previous loop, and the same amount of data would be loaded into the cache. However, the program needs only one of the eight 32-bit words loaded into cache. Even though this program performs one-eighth the additions of the previous loop, its elapsed time is roughly the same as the previous loop because the memory operations dominate performance.

While this example may seem a bit contrived, there are several situations in which non-unit strides occur quite often. First, when a FORTRAN two-dimensional array is stored in memory, successive elements in the first column are stored sequentially followed by the elements of the second column. If the array is processed with the row iteration as the inner loop, it produces a unit-stride reference pattern as follows:

```
REAL*4 A(200,200)
DO J = 1,200
  DO I = 1,200
    SUM = SUM + A(I,J)
  END DO
END DO
```

Interestingly, a FORTRAN programmer would most likely write the loop (in alphabetical order) as follows, producing a non-unit stride of 800 bytes between successive load operations:

```
REAL*4 A(200,200)
DO I = 1,200
  DO J = 1,200
    SUM = SUM + A(I,J)
  END DO
END DO
```

Because of this, some compilers can detect this suboptimal loop order and reverse the order of the loops to make best use of the memory system. As we will see in Chapter 4, however, this code transformation may produce different results, and so you may have to give the compiler "permission" to interchange these loops in this particular example (or, after reading this book, you could just code it properly in the first place).

```
while ( ptr != NULL ) ptr = ptr->next;
```

The next element that is retrieved is based on the contents of the current element. This type of loop bounces all around memory in no particular pattern. This is called *pointer chasing* and there are no good ways to improve the performance of this code.

A third pattern often found in certain types of codes is called gather (or scatter) and occurs in loops such as:

```
SUM = SUM + ARR ( IND(I) )
```

where the IND array contains offsets into the ARR array. Again, like the linked list, the exact pattern of memory references is known only at runtime when the values stored in the IND array are known. Some special-purpose systems have special hardware support to accelerate this particular operation.

2.5 Cache Organization[8]

2.5.1 Cache Organization

The process of pairing memory locations with cache lines is called *mapping*. Of course, given that a cache is smaller than main memory, you have to share the same cache lines for different memory locations. In caches, each cache line has a record of the memory address (called the *tag*) it represents and perhaps when it was last used. The tag is used to track which area of memory is stored in a particular cache line.

The way memory locations (tags) are mapped to cache lines can have a beneficial effect on the way your program runs, because if two heavily used memory locations map onto the same cache line, the miss rate will be higher than you would like it to be. Caches can be organized in one of several ways: direct mapped, fully associative, and set associative.

2.5.1.1 Direct-Mapped Cache

Direct mapping, as shown in Figure 2.2 (Figure 3-2: Many memory addresses map to the same cache line), is the simplest algorithm for deciding how memory maps onto the cache. Say, for example, that your computer has a 4-KB cache. In a direct mapped scheme, memory location 0 maps into cache location 0, as do memory locations 4K, 8K, 12K, etc. In other words, memory maps onto the cache size. Another way to think about it is to imagine a metal spring with a chalk line marked down the side. Every time around the spring, you encounter the chalk line at the same place modulo the circumference of the spring. If the spring is very long, the chalk line crosses many coils, the analog being a large memory with many locations mapping into the same cache line.

Problems occur when alternating runtime memory references in a direct-mapped cache point to the same cache line. Each reference causes a cache miss and replaces the entry just replaced, causing a lot of overhead. The popular word for this is *thrashing*. When there is lots of thrashing, a cache can be more of a liability than an asset because each cache miss requires that a cache line be refilled — an operation that moves more data than merely satisfying the reference directly from main memory. It is easy to construct a pathological case that causes thrashing in a 4-KB direct-mapped cache:

[8] This content is available online at <http://cnx.org/content/m32722/1.1/>.

Figure 3-2: Many memory addresses map to the same cache line

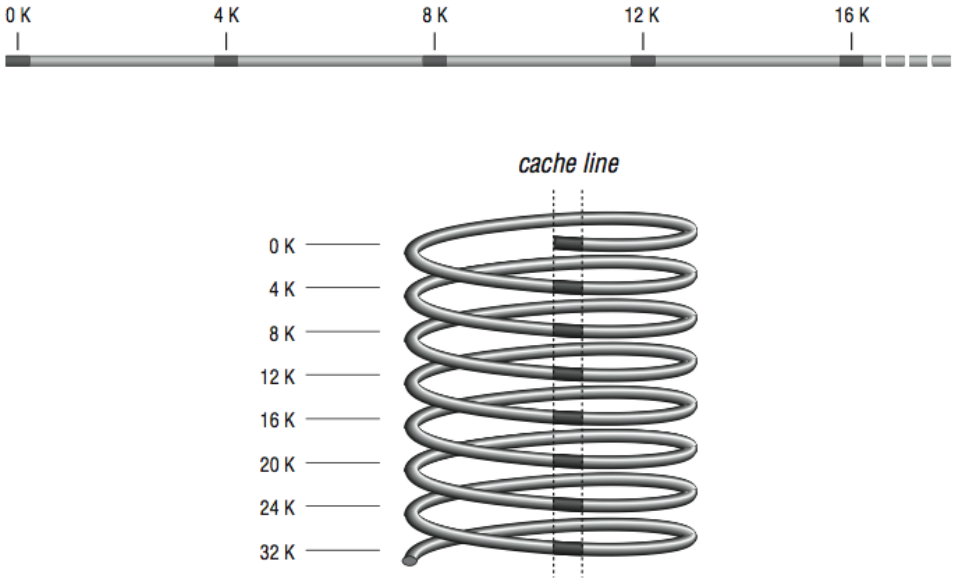


Figure 2.2

```
REAL*4 A(1024), B(1024)
COMMON /STUFF/ A,B
DO I=1,1024
  A(I) = A(I) * B(I)
END DO
END
```

The arrays A and B both take up exactly 4 KB of storage, and their inclusion together in `COMMON` assures that the arrays start exactly 4 KB apart in memory. In a 4-KB direct mapped cache, the same line that is used for A(1) is used for B(1), and likewise for A(2) and B(2), etc., so alternating references cause repeated cache misses. To fix it, you could either adjust the size of the array A, or put some other variables into `COMMON`, between them. For this reason one should generally avoid array dimensions that are close to powers of two.

2.5.1.2 Fully Associative Cache

At the other extreme from a direct mapped cache is a *fully associative cache*, where any memory location can be mapped into any cache line, regardless of memory address. Fully associative caches get their name from the type of memory used to construct them — associative memory. Associative memory is like regular memory, except that each memory cell knows something about the data it contains.

When the processor goes looking for a piece of data, the cache lines are asked all at once whether any of them has it. The cache line containing the data holds up its hand and says "I have it"; if none of them do, there is a cache miss. It then becomes a question of which cache line will be replaced with the new data. Rather than map memory locations to cache lines via an algorithm, like a direct- mapped cache, the memory system can ask the fully associative cache lines to choose among themselves which memory locations they will represent. Usually the least recently used line is the one that gets overwritten with new data. The assumption is that if the data hasn't been used in quite a while, it is least likely to be used in the future.

Fully associative caches have superior utilization when compared to direct mapped caches. It's difficult to find real-world examples of programs that will cause thrashing in a fully associative cache. The expense of fully associative caches is very high, in terms of size, price, and speed. The associative caches that do exist tend to be small.

2.5.1.3 Set-Associative Cache

Now imagine that you have two direct mapped caches sitting side by side in a single cache unit as shown in Figure 2.3 (Figure 3-3: Two-way set-associative cache). Each memory location corresponds to a particular cache line in each of the two direct-mapped caches. The one you choose to replace during a cache miss is subject to a decision about whose line was used last — the same way the decision was made in a fully associative cache except that now there are only two choices. This is called a *set-associative cache*. Set-associative caches generally come in two and four separate banks of cache. These are called *two-way* and *four-way* set associative caches, respectively. Of course, there are benefits and drawbacks to each type of cache. A set-associative cache is more immune to cache thrashing than a direct-mapped cache of the same size, because for each mapping of a memory address into a cache line, there are two or more choices where it can go. The beauty of a direct-mapped cache, however, is that it's easy to implement and, if made large enough, will perform roughly as well as a set-associative design. Your machine may contain multiple caches for several different purposes. Here's a little program for causing thrashing in a 4-KB two-way set- associative cache:

```
REAL*4 A(1024), B(1024), C(1024)
COMMON /STUFF/ A,B,C
DO I=1,1024
  A(I) = A(I) * B(I) + C(I)
END DO
END
```

Like the previous cache thrasher program, this forces repeated accesses to the same cache lines, except that now there are three variables contending for the choose set same mapping instead of two. Again, the way to fix it would be to change the size of the arrays or insert something in between them, in `COMMON`. By the way, if you accidentally arranged a program to thrash like this, it would be hard for you to detect it — aside from a feeling that the program runs a little slow. Few vendors provide tools for measuring cache misses.

Figure 3-3: Two-way set-associative cache

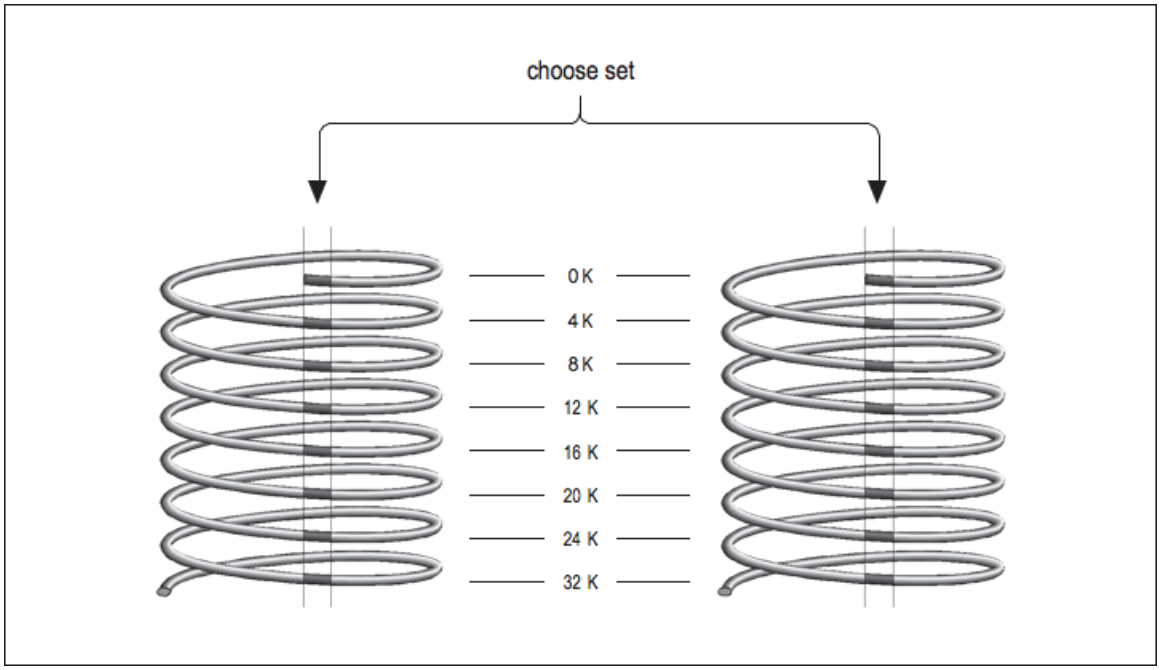


Figure 2.3

2.5.1.4 Instruction Cache

So far we have glossed over the two kinds of information you would expect to find in a cache between main memory and the CPU: instructions and data. But if you think about it, the demand for data is separate from the demand for instructions. In superscalar processors, for example, it's possible to execute an instruction that causes a data cache miss alongside other instructions that require no data from cache at all, i.e., they operate on registers. It doesn't seem fair that a cache miss on a data reference in one instruction should keep you from fetching other instructions because the cache is tied up. Furthermore, a cache depends on locality of reference between bits of data and other bits of data or instructions and other instructions, but what kind of interplay is there between instructions and data? It would seem possible for instructions to bump perfectly useful data from cache, or vice versa, with complete disregard for locality of reference.

Many designs from the 1980s used a single cache for both instructions and data. But newer designs are employing what is known as the *Harvard Memory Architecture*, where the demand for data is segregated from the demand for instructions.

Main memory is a still a single large pool, but these processors have separate data and instruction caches, possibly of different designs. By providing two independent sources for data and instructions, the aggregate rate of information coming from memory is increased, and interference between the two types of memory references is minimized. Also, instructions generally have an extremely high level of locality of reference because of the sequential nature of most programs. Because the instruction caches don't have to be particularly large to be effective, a typical architecture is to have separate L1 caches for instructions and data and to have a combined L2 cache. For example, the IBM/Motorola PowerPC 604e has separate 32-K

four-way set-associative L1 caches for instruction and data and a combined L2 cache.

2.6 Virtual Memory[9]

2.6.1 Virtual Memory

Virtual memory decouples the addresses used by the program (virtual addresses) from the actual addresses where the data is stored in memory (physical addresses). Your program sees its address space starting at 0 and working its way up to some large number, but the actual physical addresses assigned can be very different. It gives a degree of flexibility by allowing all processes to believe they have the entire memory system to themselves. Another trait of virtual memory systems is that they divide your program's memory up into *pages* — chunks. Page sizes vary from 512 bytes to 1 MB or larger, depending on the machine. Pages don't have to be allocated contiguously, though your program sees them that way. By being separated into pages, programs are easier to arrange in memory, or move portions out to disk.

2.6.1.1 Page Tables

Say that your program asks for a variable stored at location 1000. In a virtual memory machine, there is no direct correspondence between your program's idea of where location 1000 is and the physical memory systems' idea. To find where your variable is actually stored, the location has to be translated from a virtual to a physical address. The map containing such translations is called a *page table*. Each process has a several page tables associated with it, corresponding to different regions, such as program text and data segments.

To understand how address translation works, imagine the following scenario: at some point, your program asks for data from location 1000. Figure 2.4 (Figure 3-4: Virtual-to-physical address mapping) shows the steps required to complete the retrieval of this data. By choosing location 1000, you have identified which region the memory reference falls in, and this identifies which page table is involved. Location 1000 then helps the processor choose an entry within the table. For instance, if the page size is 512 bytes, 1000 falls within the second page (pages range from addresses 0–511, 512–1023, 1024–1535, etc.).

Therefore, the second table entry should hold the address of the page housing the value at location 1000.

[9] This content is available online at <http://cnx.org/content/m32728/1.1/>.

Figure 3-4: Virtual-to-physical address mapping

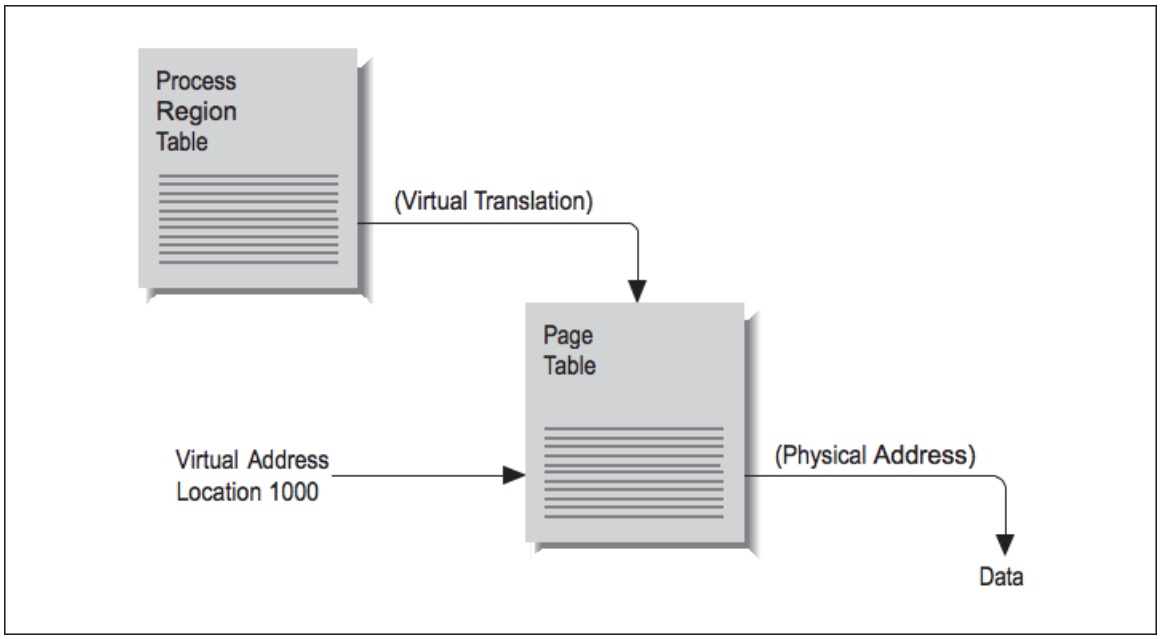


Figure 2.4

The operating system stores the page-table addresses virtually, so it's going to take a virtual-to-physical translation to locate the table in memory. One more virtual-to- physical translation, and we finally have the true address of location 1000. The memory reference can complete, and the processor can return to executing your program.

2.6.1.2 Translation Lookaside Buffer

As you can see, address translation through a page table is pretty complicated. It required two table lookups (maybe three) to locate our data. If every memory reference was that complicated, virtual memory computers would be horrible performers. Fortunately, locality of reference causes virtual address translations to group together; a program may repeat the same virtual page mapping millions of times a second. And where we have repeated use of the same data, we can apply a cache.

All modern virtual memory machines have a special cache called a *translation lookaside buffer* (TLB) for virtual-to-physical-memory-address translation. The two inputs to the TLB are an integer that identifies the program making the memory request and the virtual page requested. From the output pops a pointer to the physical page number. Virtual address in; physical address out. TLB lookups occur in parallel with instruction execution, so if the address data is in the TLB, memory references proceed quickly.

Like other kinds of caches, the TLB is limited in size. It doesn't contain enough entries to handle all the possible virtual-to-physical-address translations for all the programs that might run on your computer. Larger pools of address translations are kept out in memory, in the page tables. If your program asks for a virtual-to- physical-address translation, and the entry doesn't exist in the TLB, you suffer a *TLB miss*. The information needed may have to be generated (a new page may need to be created), or it may have to be retrieved from the page table.

The TLB is good for the same reason that other types of caches are good: it reduces the cost of memory references. But like other caches, there are pathological cases where the TLB can fail to deliver value. The

easiest case to construct is one where every memory reference your program makes causes a TLB miss:

```
REAL X(10000000)
COMMON X
DO I=0,9999
  DO J=1,10000000,10000
    SUM = SUM + X(J+I)
  END DO
END DO
```

Assume that the TLB page size for your computer is less than 40 KB. Every time through the inner loop in the above example code, the program asks for data that is 4 bytes*10,000 = 40,000 bytes away from the last reference. That is, each reference falls on a different memory page. This causes 1000 TLB misses in the inner loop, taken 1001 times, for a total of at least one million TLB misses. To add insult to injury, each reference is guaranteed to cause a data cache miss as well. Admittedly, no one would start with a loop like the one above. But presuming that the loop was any good to you at all, the restructured version in the code below would cruise through memory like a warm knife through butter:

```
REAL X(10000000)
COMMON X
DO I=1,10000000
  SUM = SUM + X(I)
END DO
```

The revised loop has unit stride, and TLB misses occur only every so often. Usually it is not necessary to explicitly tune programs to make good use of the TLB. Once a program is tuned to be "cache-friendly," it nearly always is tuned to be TLB friendly.

Because there is a performance benefit to keeping the TLB very small, the TLB entry often contains a length field. A single TLB entry can be over a megabyte in length and can be used to translate addresses stored in multiple virtual memory pages.

2.6.1.3 Page Faults

A page table entry also contains other information about the page it represents, including flags to tell whether the translation is valid, whether the associated page can be modified, and some information describing how new pages should be initialized. References to pages that aren't marked valid are called *page faults*.

Taking a worst-case scenario, say that your program asks for a variable from a particular memory location. The processor goes to look for it in the cache and finds it isn't there (cache miss), which means it must be loaded from memory. Next it goes to the TLB to find the physical location of the data in memory and finds there is no TLB entry (a TLB miss). Then it tries consulting the page table (and refilling the TLB), but finds that either there is no entry for your particular page or that the memory page has been shipped to disk (both are page faults). Each step of the memory hierarchy has shrugged off your request. A new page will have to be created in memory and possibly, depending on the circumstances, refilled from disk.

Although they take a lot of time, page faults aren't errors. Even under optimal conditions every program suffers some number of page faults. Writing a variable for the first time or calling a subroutine that has

never been called can cause a page fault. This may be surprising if you have never thought about it before. The illusion is that your entire program is present in memory from the start, but some portions may never be loaded. There is no reason to make space for a page whose data is never referenced or whose instructions are never executed. Only those pages that are required to run the job get created or pulled in from the disk.[10]

The pool of physical memory pages is limited because physical memory is limited, so on a machine where many programs are lobbying for space, there will be a higher number of page faults. This is because physical memory pages are continually being recycled for other purposes. However, when you have the machine to yourself, and memory is less in demand, allocated pages tend to stick around for a while. In short, you can expect fewer page faults on a quiet machine. One trick to remember if you ever end up working for a computer vendor: always run short benchmarks twice. On some systems, the number of page faults will go down. This is because the second run finds pages left in memory by the first, and you won't have to pay for page faults again.[11]

Paging space (swap space) on the disk is the last and slowest piece of the memory hierarchy for most machines. In the worst-case scenario we saw how a memory reference could be pushed down to slower and slower performance media before finally being satisfied. If you step back, you can view the disk paging space as having the same relationship to main memory as main memory has to cache. The same kinds of optimizations apply too, and locality of reference is important. You can run programs that are larger than the main memory system of your machine, but sometimes at greatly decreased performance. When we look at memory optimizations in Chapter 8, we will concentrate on keeping the activity in the fastest parts of the memory system and avoiding the slow parts.

2.7 Improving Memory Performance[12]

2.7.1 Improving Memory Performance

Given the importance, in the area of high performance computing, of the performance of a computer's memory subsystem, many techniques have been used to improve the performance of the memory systems of computers. The two attributes of memory system performance are generally *bandwidth* and *latency*. Some memory system design changes improve one at the expense of the other, and other improvements positively impact both bandwidth and latency. Bandwidth generally focuses on the best possible steady-state transfer rate of a memory system. Usually this is measured while running a long unit-stride loop reading or reading and writing memory.[13] Latency is a measure of the worst-case performance of a memory system as it moves a small amount of data such as a 32- or 64-bit word between the processor and memory. Both are important because they are an important part of most high performance applications.

Because memory systems are divided into components, there are different bandwidth and latency figures between different components as shown in Figure 2.5 (Figure 3-5: Simple Memory System). The bandwidth rate between a cache and the CPU will be higher than the bandwidth between main memory and the cache, for instance. There may be several caches and paths to memory as well. Usually, the peak memory bandwidth quoted by vendors is the speed between the data cache and the processor.

In the rest of this section, we look at techniques to improve latency, bandwidth, or both.

2.7.1.1 Large Caches

As we mentioned at the start of this chapter, the disparity between CPU speeds and memory is growing. If you look closely, you can see vendors innovating in several ways. Some workstations are being offered with 4- MB data caches! This is larger than the main memory systems of machines just a few years ago. With a large enough cache, a small (or even moderately large) data set can fit completely inside and get incredibly

[10] The term for this is demand paging.
[11] Text pages are identified by the disk device and block number from which they came.
[12] This content is available online at <http://cnx.org/content/m32736/1.1/>.
[13] See the STREAM section in Chapter 15 for measures of memory bandwidth.

good performance. Watch out for this when you are testing new hardware. When your program grows too large for the cache, the performance may drop off considerably, perhaps by a factor of 10 or more, depending on the memory access patterns. Interestingly, an increase in cache size on the part of vendors can render a benchmark obsolete.

Figure 3-5: Simple Memory System

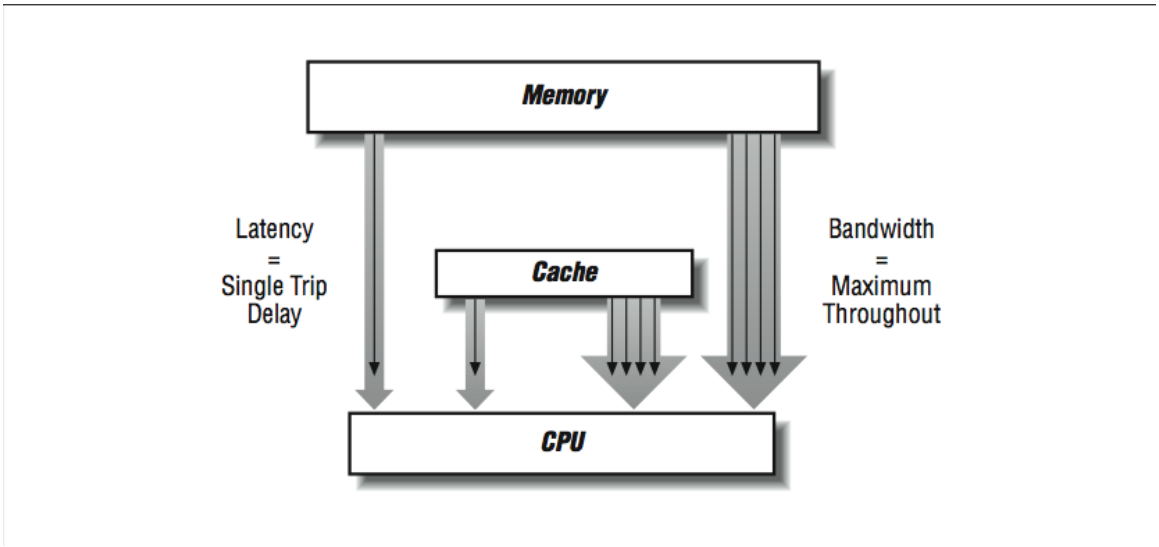


Figure 2.5

Up to 1992, the Linpack 100×100 benchmark was probably the single most- respected benchmark to determine the average performance across a wide range of applications. In 1992, IBM introduced the IBM RS-6000 which had a cache large enough to contain the entire 100×100 matrix for the duration of the benchmark. For the first time, a workstation had performance on this benchmark on the same order of supercomputers. In a sense, with the entire data structure in a SRAM cache, the RS-6000 was operating like a Cray vector supercomputer. The problem was that the Cray could maintain and improve the performance for a 120×120 matrix, whereas the RS-6000 suffered a significant performance loss at this increased matrix size. Soon, all the other workstation vendors introduced similarly large caches, and the 100×100 Linpack benchmark ceased to be useful as an indicator of average application performance.

2.7.1.2 Wider Memory Systems

Consider what happens when a cache line is refilled from memory: consecutive memory locations from main memory are read to fill consecutive locations within the cache line. The number of bytes transferred depends on how big the line is — anywhere from 16 bytes to 256 bytes or more. We want the refill to proceed quickly because an instruction is stalled in the pipeline, or perhaps the processor is waiting for more instructions. In Figure 2.6 (Figure 3-6: Narrow memory system), if we have two DRAM chips that provide us with 4 bits of data every 100 ns (remember cycle time), a cache fill of a 16-byte line takes 1600 ns.

Figure 3-6: Narrow memory system

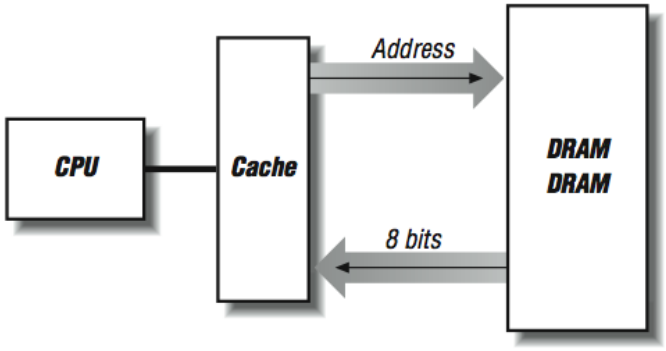


Figure 2.6

One way to make the cache-line fill operation faster is to "widen" the memory system as shown in Figure 2.7 (Figure 3-7: Wide memory system). Instead of having two rows of DRAMs, we create multiple rows of DRAMs. Now on every 100-ns cycle, we get 32 contiguous bits, and our cache-line fills are four times faster.

Figure 3-7: Wide memory system

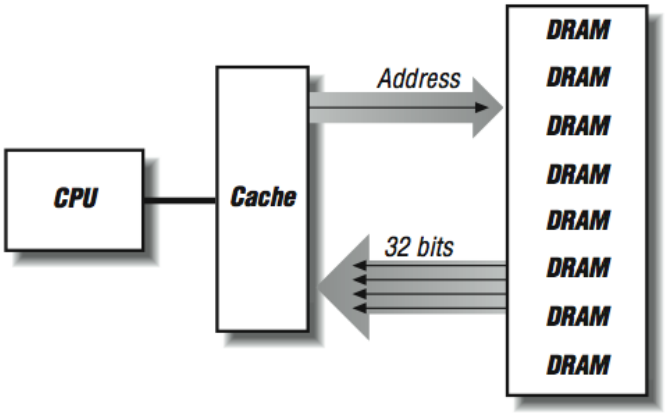


Figure 2.7

We can improve the performance of a memory system by increasing the width of the memory system up to the length of the cache line, at which time we can fill the entire line in a single memory cycle. On the SGI Power Challenge series of systems, the memory width is 256 bits. The downside of a wider memory system is that DRAMs must be added in multiples. In many modern workstations and personal computers, memory is expanded in the form of single inline memory modules (SIMMs). SIMMs currently are either 30-, 72-, or 168-pin modules, each of which is made up of several DRAM chips ready to be installed into a memory sub-system.

2.7.1.3 Bypassing Cache

It's interesting that we have spent nearly an entire chapter on how great a cache is for high performance computers, and now we are going to bypass the cache to improve performance. As mentioned earlier, some types of processing result in non-unit strides (or bouncing around) through memory. These types of memory reference patterns bring out the worst-case behavior in cache-based architectures. It is these reference patterns that see improved performance by bypassing the cache. Inability to support these types of computations remains an area where traditional supercomputers can significantly outperform high-speed RISC processors. For this reason, RISC processors that are serious about number crunching may have special instructions that bypass data cache memory; the data are transferred directly between the processor and the main memory system.[14] In Figure 2.8 (Figure 3-8: Bypassing cache) we have four banks of SIMMs that can do cache fills at 128 bits per 100 ns memory cycle. Remember that the data is available after 50 ns but we can't get more data until the DRAMs refresh 50–60 ns later. However, if we are doing 32-bit non-unit-stride loads and have the capability to bypass cache, each load will be satisfied from one of the four SIMMs in 50 ns. While that SIMM refreshed, another load can occur from any of the other three SIMMs in 50 ns. In a random mix of non-unit loads there is a 75% chance that the next load will fall on a "fresh" DRAM. If the load falls on a bank while it is refreshing, it simply has to wait until the refresh completes.

A further advantage of bypassing cache is that the data doesn't need to be moved through the SRAM cache. This operation can add from 10–50 ns to the load time for a single word. This also avoids invalidating the contents of an entire cache line in the cache.

Adding cache bypass, increasing memory-system widths, and adding banks increases the cost of a memory system. Computer-system vendors make an economic choice as to how many of these techniques they need to apply to get sufficient performance for their particular processor and system. Hence, as processor speed increases, vendors must add more of these memory system features to their commodity systems to maintain a balance between processor and memory-system speed.

[14] By the way, most machines have uncached memory spaces for process synchronization and I/O device registers. However, memory references to these locations bypass the cache because of the address chosen, not necessarily because of the instruction chosen.

Figure 3-8: Bypassing cache

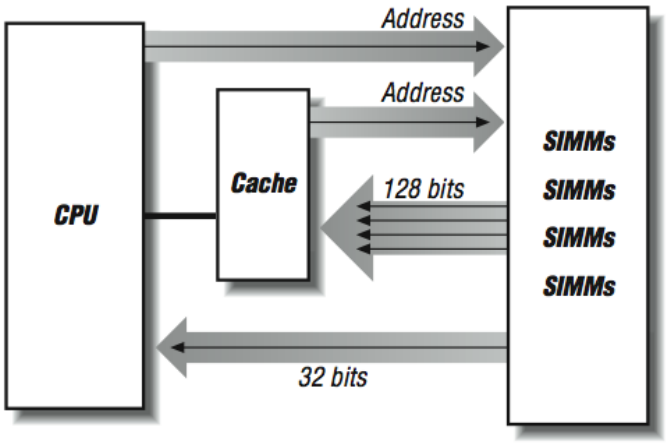


Figure 2.8

2.7.1.4 Interleaved and Pipelined Memory Systems

Vector supercomputers, such as the CRAY Y/MP and the Convex C3, are machines that depend on multi-banked memory systems for performance. The C3, in particular, has a memory system with up to 256-way interleaving. Each interleave (or bank) is 64 bits wide. This is an expensive memory system to build, but it has some very nice performance characteristics. Having a large number of banks helps to reduce the chances of repeated access to the same memory bank. If you do hit the same bank twice in a row, however, the penalty is a delay of nearly 300 ns — a long time for a machine with a clock speed of 16 ns. So when things go well, they go very well.

However, having a large number of banks alone is not sufficient to feed a 16-ns processor using 50 ns DRAM. In addition to interleaving, the memory subsystem also needs to be pipelined. That is, the CPU must begin the second, third, and fourth load before the CPU has received the results of the first load as shown in Figure 2.9 (Figure 3-9: Multibanked memory system). Then each time it receives the results from bank "n," it must start the load from bank "n+4" to keep the pipeline fed. This way, after a brief startup delay, loads complete every 16 ns and so the memory system appears to operate at the clock rate of the CPU. This pipelined memory approach is facilitated by the 128-element vector registers in the C3 processor.

Using gather/scatter hardware, non-unit-stride operations can also be pipelined. The only difference for non-unit-stride operations is that the banks are not accessed in sequential order. With a random pattern of memory references, it's possible to reaccess a memory bank before it has completely refreshed from a previous access. This is called a *bank stall*.

Figure 3-9: Multibanked memory system

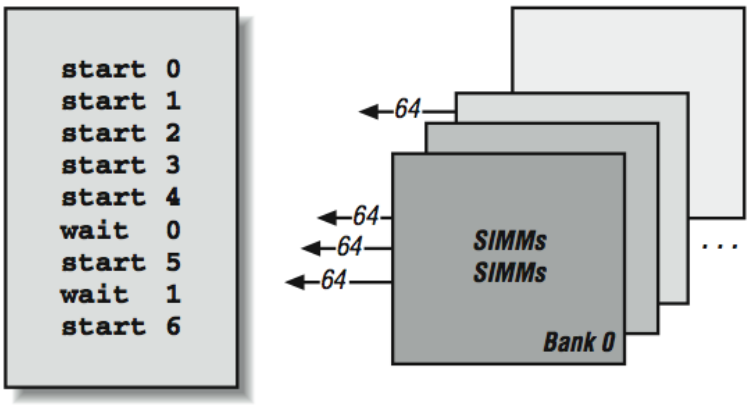


Figure 2.9

Different access patterns are subject to bank stalls of varying severity. For instance, accesses to every fourth word in an eight-bank memory system would also be subject to bank stalls, though the recovery would occur sooner. References to every second word might not experience bank stalls at all; each bank may have recovered by the time its next reference comes around; it depends on the relative speeds of the processor and memory system. Irregular access patterns are sure to encounter some bank stalls.

In addition to the bank stall hazard, single-word references made directly to a multibanked memory system carry a greater latency than those of (successfully) cached memory accesses. This is because references are going out to memory that is slower than cache, and there may be additional address translation steps as well. However, banked memory references are pipelined. As long as references are started well enough in advance, several pipelined, multibanked references can be in flight at one time, giving you good throughput.

The CDC-205 system performed vector operations in a memory-to-memory fashion using a set of explicit memory pipelines. This system had superior performance for very long unit-stride vector computations. A single instruction could perform 65,000 computations using three memory pipes.

2.7.1.5 Software Managed Caches

Here's an interesting thought: if a vector processor can plan far enough in advance to start a memory pipe, why can't a RISC processor start a cache-fill before it really needs the data in those same situations? In a way, this is priming the cache to hide the latency of the cache-fill. If this could be done far enough in advance, it would appear that all memory references would operate at the speed of the cache.

This concept is called prefetching and it is supported using a special prefetch instruction available on many RISC processors. A prefetch instruction operates just like a standard load instruction, except that the processor doesn't wait for the cache to fill before the instruction completes. The idea is to prefetch far enough ahead of the computation to have the data ready in cache by the time the actual computation occurs. The following is an example of how this might be used:

```
DO I=1,1000000,8
  PREFETCH(ARR(I+8))
  DO J=0,7
    SUM=SUM+ARR(I+J)
  END DO
END DO
```

This is not the actual FORTRAN. Prefetching is usually done in the assembly code generated by the compiler when it detects that you are stepping through the array using a fixed stride. The compiler typically estimate how far ahead you should be prefetching. In the above example, if the cache-fills were particularly slow, the value 8 in I+8 could be changed to 16 or 32 while the other values changed accordingly.

In a processor that could only issue one instruction per cycle, there might be no payback to a prefetch instruction; it would take up valuable time in the instruction stream in exchange for an uncertain benefit. On a superscalar processor, however, a cache hint could be mixed in with the rest of the instruction stream and issued alongside other, real instructions. If it saved your program from suffering extra cache misses, it would be worth having.

2.7.1.6 Post-RISC Effects on Memory References

Memory operations typically access the memory during the execute phase of the pipeline on a RISC processor. On the post-RISC processor, things are no different than on a RISC processor except that many loads can be half finished at any given moment. On some current processors, up to 28 memory operations may be active with 10 waiting for off-chip memory to arrive. This is an excellent way to compensate for slow memory latency compared to the CPU speed. Consider the following loop:

```
        LOADI     R6,10000    Set the Iterations
        LOADI     R5,0        Set the index variable
LOOP:   LOAD      R1,R2(R5)   Load a value from memory
        INCR      R1          Add one to R1
        STORE     R1,R3(R5)   Store the incremented value back to memory
        INCR      R5          Add one to R5
        COMPARE   R5,R6       Check for loop termination
        BLT       LOOP        Branch if R5 < R6 back to LOOP
```

In this example, assume that it take 50 cycles to access memory. When the fetch/ decode puts the first load into the instruction reorder buffer (IRB), the load starts on the next cycle and then is suspended in the execute phase. However, the rest of the instructions are in the IRB. The INCR R1 must wait for the load and the STORE must also wait. However, by using a rename register, the INCR R5, COMPARE, and BLT can all be computed, and the fetch/decode goes up to the top of the loop and sends another load into the IRB for the next memory location that will have to wait. This looping continues until about 10 iterations of the loop are in the IRB. Then the first load actually shows up from memory and the INCR R1 and STORE from the first iteration begins executing. Of course the store takes a while, but about that time the second load finishes, so there is more work to do and so on...

Like many aspects of computing, the post-RISC architecture, with its out-of-order and speculative execution, optimizes memory references. The post-RISC processor dynamically unrolls loops at execution time to compensate for memory subsystem delay. Assuming a pipelined multibanked memory system that can have multiple memory operations started before any complete (the HP PA-8000 can have 10 off- chip memory operations in flight at one time), the processor continues to dispatch memory operations until those operations begin to complete.

Unlike a vector processor or a prefetch instruction, the post-RISC processor does not need to anticipate the precise pattern of memory references so it can carefully control the memory subsystem. As a result, the post-RISC processor can achieve peak performance in a far-wider range of code sequences than either vector processors or in-order RISC processors with prefetch capability.

This implicit tolerance to memory latency makes the post-RISC processors ideal for use in the scalable shared-memory processors of the future, where the memory hierarchy will become even more complex than current processors with three levels of cache and a main memory.

Unfortunately, the one code segment that doesn't benefit significantly from the post-RISC architecture is the linked-list traversal. This is because the next address is never known until the previous load is completed so all loads are fundamentally serialized.

2.7.1.7 Dynamic RAM Technology Trends

Much of the techniques in this section have focused on how to deal with the imperfections of the dynamic RAM chip (although when your clock rate hits 300–600 MHz or 3–2 ns, even SRAM starts to look pretty slow). It's clear that the demand for more and more RAM will continue to increase, and gigabits and more DRAM will fit on a single chip. Because of this, significant work is underway to make new super-DRAMs faster and more tuned to the extremely fast processors of the present and the future. Some of the technologies are relatively straightforward, and others require a major redesign of the way that processors and memories are manufactured.

Some DRAM improvements include:

- Fast page mode DRAM
- Extended data out RAM (EDO RAM)
- Synchronous DRAM (SDRAM)
- RAMBUS
- Cached DRAM (CDRAM)

Fast *page mode* DRAM saves time by allowing a mode in which the entire address doesn't have to be re-clocked into the chip for each memory operation. Instead, there is an assumption that the memory will be accessed sequentially (as in a cache-line fill), and only the low-order bits of the address are clocked in for successive reads or writes.

EDO RAM is a modification to output buffering on page mode RAM that allows it to operate roughly twice as quickly for operations other than refresh.

Synchronous DRAM is synchronized using an external clock that allows the cache and the DRAM to coordinate their operations. Also, SDRAM can pipeline the retrieval of multiple memory bits to improve overall throughput.

RAMBUS is a proprietary technology capable of 500 MB/sec data transfer. RAMBUS uses significant logic within the chip and operates at higher power levels than typical DRAM.

Cached DRAM combines a SRAM cache on the same chip as the DRAM. This tightly couples the SRAM and DRAM and provides performance similar to SRAM devices with all the limitations of any cache architecture. One advantage of the CDRAM approach is that the amount of cache is increased as the amount of DRAM is increased. Also when dealing with memory systems with a large number of interleaves, each interleave has its own SRAM to reduce latency, assuming the data requested was in the SRAM.

An even more advanced approach is to integrate the processor, SRAM, and DRAM onto a single chip clocked at say 5 GHz, containing 128 MB of data. Understandably, there is a wide range of technical problems to solve before this type of component is widely available for $200 — but it's not out of the question. The manufacturing processes for DRAM and processors are already beginning to converge in some ways (RAMBUS). The biggest performance problem when we have this type of system will be, "What to do if you need 160 MB?"

2.8 Closing Notes[15]

2.8.1 Closing Notes

They say that the computer of the future will be a good memory system that just happens to have a CPU attached. As high performance microprocessor systems take over as *the* high performance computing engines, the problem of a cache-based memory system that uses DRAM for main memory must be solved. There are many architecture and technology efforts underway to transform workstation and personal computer memories to be as capable as supercomputer memories.

As CPU speed increases faster than memory speed, you will need the techniques in this book. Also, as you move into multiple processors, memory problems don't get better; usually they get worse. With many hungry processors always ready for more data, a memory subsystem can become extremely strained.

With just a little skill, we can often restructure memory accesses so that they play to your memory system's strengths instead of its weaknesses.

2.9 Exercises[16]

2.9.1 Exercises

Exercise 2.1
The following code segment traverses a pointer chain:

```
while ((p = (char *) *p) != NULL);
```

How will such a code interact with the cache if all the references fall within a small portion of memory? How will the code interact with the cache if references are stretched across many megabytes?

Exercise 2.2
How would the code in Exercise 2.1 behave on a multibanked memory system that has no cache?

Exercise 2.3
A long time ago, people regularly wrote self-modifying code — programs that wrote into instruction memory and changed their own behavior. What would be the implications of self-modifying code on a machine with a Harvard memory architecture?

Exercise 2.4
Assume a memory architecture with an L1 cache speed of 10 ns, L2 speed of 30 ns, and memory speed of 200 ns. Compare the average memory system performance with (1) L1 80%, L2 10%, and memory 10%; and (2) L1 85% and memory 15%.

Exercise 2.5
On a computer system, run loops that process arrays of varying length from 16 to 16 million:

```
ARRAY(I) = ARRAY(I) + 3
```

How does the number of additions per second change as the array length changes? Experiment with `REAL*4`, `REAL*8`, `INTEGER*4`, and `INTEGER*8`.

Which has more significant impact on performance: larger array elements or integer versus floating-point? Try this on a range of different computers.

Exercise 2.6
Create a two-dimensional array of 1024×1024. Loop through the array with rows as the inner loop and then again with columns as the inner loop. Perform a simple operation on each element. Do the loops perform differently? Why? Experiment with different dimensions for the array and see the performance impact.

[15] This content is available online at <http://cnx.org/content/m32690/1.1/>.
[16] This content is available online at <http://cnx.org/content/m32698/1.1/>.

Exercise 2.7
Write a program that repeatedly executes timed loops of different sizes to determine the cache size for your system.

Chapter 3

Floating-Point Numbers

3.1 Introduction[1]

3.1.1 Floating-Point Numbers

Often when we want to make a point that nothing is sacred, we say, "one plus one does not equal two." This is designed to shock us and attack our fundamental assumptions about the nature of the universe. Well, in this chapter on floating- point numbers, we will learn that "$0.1 + 0.1$ does not always equal 0.2" when we use floating-point numbers for computations.

In this chapter we explore the limitations of floating-point numbers and how you as a programmer can write code to minimize the effect of these limitations. This chapter is just a brief introduction to a significant field of mathematics called *numerical analysis*.

3.2 Reality[2]

3.2.1 Reality

The real world is full of real numbers. Quantities such as distances, velocities, masses, angles, and other quantities are all real numbers.[3] A wonderful property of real numbers is that they have unlimited accuracy. For example, when considering the ratio of the circumference of a circle to its diameter, we arrive at a value of 3.141592.... The decimal value for *pi* does not terminate. Because real numbers have unlimited accuracy, even though we can't write it down, *pi* is still a real number. Some real numbers are rational numbers because they can be represented as the ratio of two integers, such as $1/3$. Not all real numbers are rational numbers. Not surprisingly, those real numbers that aren't rational numbers are called irrational. You probably would not want to start an argument with an irrational number unless you have a lot of free time on your hands.

Unfortunately, on a piece of paper, or in a computer, we don't have enough space to keep writing the digits of *pi*. So what do we do? We decide that we only need so much accuracy and round real numbers to a certain number of digits. For example, if we decide on four digits of accuracy, our approximation of *pi* is 3.142. Some state legislature attempted to pass a law that *pi* was to be three. While this is often cited as evidence for the IQ of governmental entities, perhaps the legislature was just suggesting that we only need one digit of accuracy for *pi*. Perhaps they foresaw the need to save precious memory space on computers when representing real numbers.

[1] This content is available online at <http://cnx.org/content/m32739/1.1/>.

[2] This content is available online at <http://cnx.org/content/m32741/1.1/>.

[3] In high performance computing we often simulate the real world, so it is somewhat ironic that we use simulated real numbers (floating-point) in those simulations of the real world.

3.3 Representation[4]

3.3.1 Representation

Given that we cannot perfectly represent real numbers on digital computers, we must come up with a compromise that allows us to approximate real numbers.[5] There are a number of different ways that have been used to represent real numbers. The challenge in selecting a representation is the trade-off between space and accuracy and the tradeoff between speed and accuracy. In the field of high performance computing we generally expect our processors to produce a floating- point result every 600-MHz clock cycle. It is pretty clear that in most applications we aren't willing to drop this by a factor of 100 just for a little more accuracy. Before we discuss the format used by most high performance computers, we discuss some alternative (albeit slower) techniques for representing real numbers.

3.3.1.1 Binary Coded Decimal

In the earliest computers, one technique was to use binary coded decimal (BCD). In BCD, each base-10 digit was stored in four bits. Numbers could be arbitrarily long with as much precision as there was memory:

```
    123.45
0001 0010 0011 0100 0101
```

This format allows the programmer to choose the precision required for each variable. Unfortunately, it is difficult to build extremely high-speed hardware to perform arithmetic operations on these numbers. Because each number may be far longer than 32 or 64 bits, they did not fit nicely in a register. Much of the floating-point operations for BCD were done using loops in microcode. Even with the flexibility of accuracy on BCD representation, there was still a need to round real numbers to fit into a limited amount of space.

Another limitation of the BCD approach is that we store a value from 0–9 in a four-bit field. This field is capable of storing values from 0–15 so some of the space is wasted.

3.3.1.2 Rational Numbers

One intriguing method of storing real numbers is to store them as rational numbers. To briefly review mathematics, rational numbers are the subset of real numbers that can be expressed as a ratio of integer numbers. For example, 22/7 and 1/2 are rational numbers. Some rational numbers, such as 1/2 and 1/10, have perfect representation as base-10 decimals, and others, such as 1/3 and 22/7, can only be expressed as infinite-length base-10 decimals. When using rational numbers, each real number is stored as two integer numbers representing the numerator and denominator. The basic fractional arithmetic operations are used for addition, subtraction, multiplication, and division, as shown in Figure 3.1 (Figure 4-1: Rational number mathematics).

---

[4] This content is available online at <http://cnx.org/content/m32772/1.1/>.

[5] Interestingly, analog computers have an easier time representing real numbers. Imagine a "water- adding" analog computer which consists of two glasses of water and an empty glass. The amount of water in the two glasses are perfectly represented real numbers. By pouring the two glasses into a third, we are adding the two real numbers perfectly (unless we spill some), and we wind up with a real number amount of water in the third glass. The problem with analog computers is knowing just how much water is in the glasses when we are all done. It is also problematic to perform 600 million additions per second using this technique without getting pretty wet. Try to resist the temptation to start an argument over whether quantum mechanics would cause the real numbers to be rational numbers. And don't point out the fact that even digital computers are really analog computers at their core. I am trying to keep the focus on floating-point values, and you keep drifting away!

Figure 4-1: Rational number mathematics

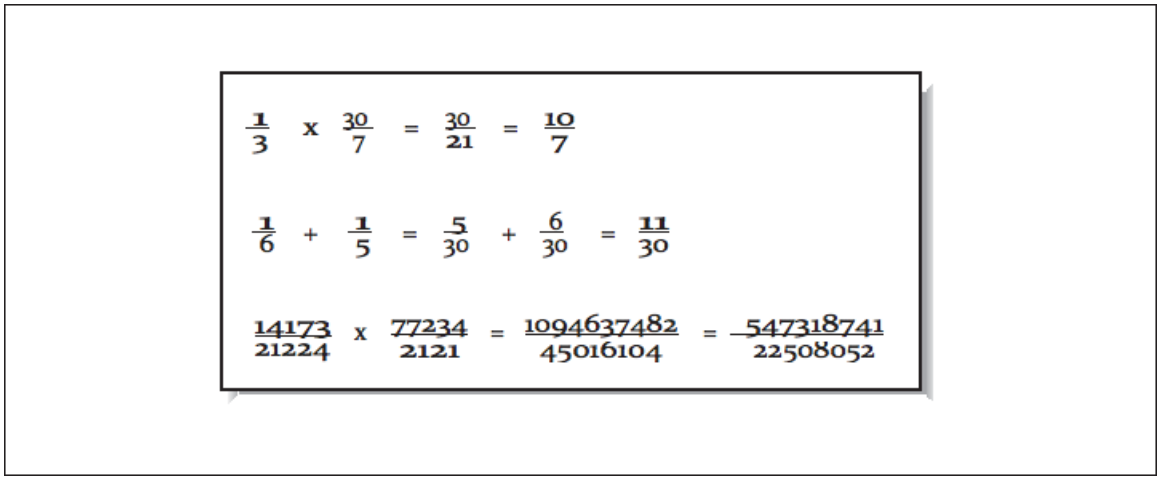


Figure 3.1

The limitation that occurs when using rational numbers to represent real numbers is that the size of the numerators and denominators tends to grow. For each addition, a common denominator must be found. To keep the numbers from becoming extremely large, during each operation, it is important to find the *greatest common divisor* (GCD) to reduce fractions to their most compact representation. When the values grow and there are no common divisors, either the large integer values must be stored using dynamic memory or some form of approximation must be used, thus losing the primary advantage of rational numbers.

For mathematical packages such as Maple or Mathematica that need to produce exact results on smaller data sets, the use of rational numbers to represent real numbers is at times a useful technique. The performance and storage cost is less significant than the need to produce exact results in some instances.

3.3.1.3 Fixed Point

If the desired number of decimal places is known in advance, it's possible to use fixed-point representation. Using this technique, each real number is stored as a scaled integer. This solves the problem that base-10 fractions such as 0.1 or 0.01 cannot be perfectly represented as a base-2 fraction. If you multiply 110.77 by 100 and store it as a scaled integer 11077, you can perfectly represent the base-10 fractional part (0.77). This approach can be used for values such as money, where the number of digits past the decimal point is small and known.

However, just because all numbers can be accurately represented it doesn't mean there are not errors with this format. When multiplying a fixed-point number by a fraction, you get digits that can't be represented in a fixed-point format, so some form of rounding must be used. For example, if you have $125.87 in the bank at 4% interest, your interest amount would be $5.0348. However, because your bank balance only has two digits of accuracy, they only give you $5.03, resulting in a balance of $130.90. Of course you probably have heard many stories of programmers getting rich depositing many of the remaining 0.0048 amounts into their own account. My guess is that banks have probably figured that one out by now, and the bank keeps the money for itself. But it does make one wonder if they round or truncate in this type of calculation.[6]

[6] Perhaps banks round this instead of truncating, knowing that they will always make it up in teller machine fees.

3.3.1.4 Mantissa/Exponent

The floating-point format that is most prevalent in high performance computing is a variation on scientific notation. In scientific notation the real number is represented using a mantissa, base, and exponent: 6.02×10^{23}.

The mantissa typically has some fixed number of places of accuracy. The mantissa can be represented in base 2, base 16, or BCD. There is generally a limited range of exponents, and the exponent can be expressed as a power of 2, 10, or 16.

The primary advantage of this representation is that it provides a wide overall range of values while using a fixed-length storage representation. The primary limitation of this format is that the difference between two successive values is not uniform. For example, assume that you can represent three base-10 digits, and your exponent can range from –10 to 10. For numbers close to zero, the "distance" between successive numbers is very small. For the number 1.72×10^{-10}, the next larger number is 1.73×10^{-10}. The distance between these two "close" small numbers is 0.000000000001. For the number 6.33×10^{10}, the next larger number is 6.34×10^{10}. The distance between these "close" large numbers is 100 million.

In Figure 3.2 (Figure 4-2: Distance between successive floating-point numbers), we use two base-2 digits with an exponent ranging from –1 to 1.

Figure 4-2: Distance between successive floating-point numbers

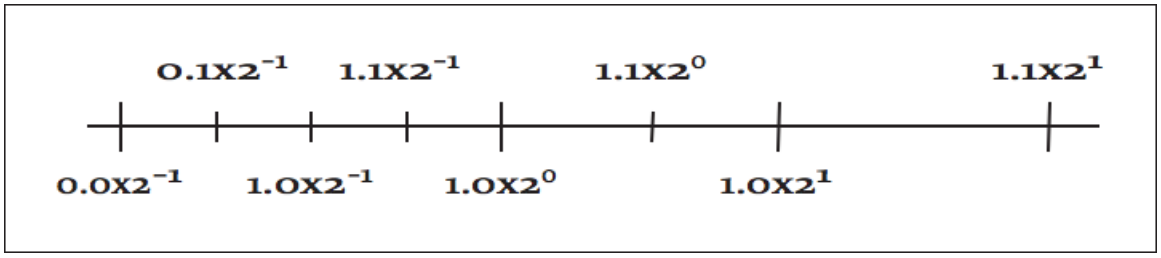


Figure 3.2

There are multiple equivalent representations of a number when using scientific notation:
6.00×10^5
0.60×10^6
0.06×10^7

By convention, we shift the mantissa (adjust the exponent) until there is exactly one nonzero digit to the left of the decimal point. When a number is expressed this way, it is said to be "normalized." In the above list, only 6.00×10^5 is normalized. Figure 3.3 (Figure 4-3: Normalized floating-point numbers) shows how some of the floating-point numbers from Figure 3.2 (Figure 4-2: Distance between successive floating-point numbers) are not normalized.

While the mantissa/exponent has been the dominant floating-point approach for high performance computing, there were a wide variety of specific formats in use by computer vendors. Historically, each computer vendor had their own particular format for floating-point numbers. Because of this, a program executed on several different brands of computer would generally produce different answers. This invariably led to heated discussions about which system provided the right answer and which system(s) were generating meaningless results.[7]

[7] Interestingly, there was an easy answer to the question for many programmers. Generally they trusted the results from the computer they used to debug the code and dismissed the results from other computers as garbage.

Figure 4-3: Normalized floating-point numbers

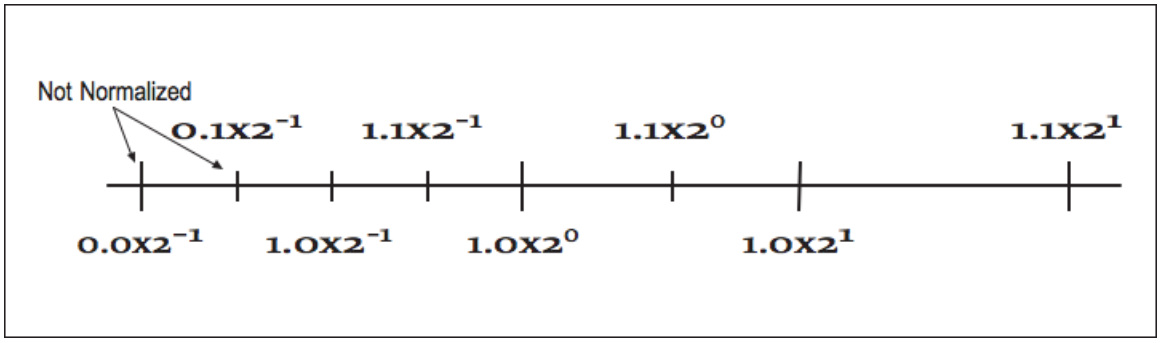


Figure 3.3

When storing floating-point numbers in digital computers, typically the mantissa is normalized, and then the mantissa and exponent are converted to base-2 and packed into a 32- or 64-bit word. If more bits were allocated to the exponent, the overall range of the format would be increased, and the number of digits of accuracy would be decreased. Also the base of the exponent could be base-2 or base-16. Using 16 as the base for the exponent increases the overall range of exponents, but because normalization must occur on four-bit boundaries, the available digits of accuracy are reduced on the average. Later we will see how the IEEE 754 standard for floating-point format represents numbers.

3.4 Effects of Floating-Point Representation[8]

3.4.1 Effects of Floating-Point Representation

One problem with the mantissa/base/exponent representation is that not all base-10 numbers can be expressed perfectly as a base-2 number. For example, 1/2 and 0.25 can be represented perfectly as base-2 values, while 1/3 and 0.1 produce infinitely repeating base-2 decimals. These values must be rounded to be stored in the floating-point format. With sufficient digits of precision, this generally is not a problem for computations. However, it does lead to some anomalies where algebraic rules do not appear to apply. Consider the following example:

```
REAL*4 X,Y
X = 0.1
Y = 0
DO I=1,10
  Y = Y + X
ENDDO
IF ( Y .EQ. 1.0 ) THEN
  PRINT *,'Algebra is truth'
ELSE
```

[8]This content is available online at <http://cnx.org/content/m32755/1.1/>.

```
  PRINT *,'Not here'
ENDIF
PRINT *,1.0-Y
END
```

At first glance, this appears simple enough. Mathematics tells us ten times 0.1 should be one. Unfortunately, because 0.1 cannot be represented exactly as a base-2 decimal, it must be rounded. It ends up being rounded down to the last bit. When ten of these slightly smaller numbers are added together, it does not quite add up to 1.0. When X and Y are `REAL*4`, the difference is about 10^{-7}, and when they are `REAL*8`, the difference is about 10^{-16}.

One possible method for comparing computed values to constants is to subtract the values and test to see how close the two values become. For example, one can rewrite the test in the above code to be:

```
IF ( ABS(1.0-Y).LT. 1E-6) THEN
  PRINT *,'Close enough for government work'
ELSE
  PRINT *,'Not even close'
ENDIF
```

The type of the variables in question and the expected error in the computation that produces Y determines the appropriate value used to declare that two values are close enough to be declared equal.

Another area where inexact representation becomes a problem is the fact that algebraic inverses do not hold with all floating-point numbers. For example, using `REAL*4`, the value `(1.0/X) * X` does not evaluate to 1.0 for 135 values of X from one to 1000. This can be a problem when computing the inverse of a matrix using LU-decomposition. LU-decomposition repeatedly does division, multiplication, addition, and subtraction. If you do the straightforward LU-decomposition on a matrix with integer coefficients that has an integer solution, there is a pretty good chance you won't get the exact solution when you run your algorithm. Discussing techniques for improving the accuracy of matrix inverse computation is best left to a numerical analysis text.

3.5 More Algebra That Doesn't Work[9]

3.5.1 More Algebra That Doesn't Work

While the examples in the proceeding section focused on the limitations of multiplication and division, addition and subtraction are not, by any means, perfect. Because of the limitation of the number of digits of precision, certain additions or subtractions have no effect. Consider the following example using `REAL*4` with 7 digits of precision:

```
X = 1.25E8
Y = X + 7.5E-3
IF ( X.EQ.Y ) THEN
  PRINT *,'Am I nuts or what?'
```

[9] This content is available online at <http://cnx.org/content/m32754/1.1/>.

```
ENDIF
```

While both of these numbers are precisely representable in floating-point, adding them is problematic. Prior to adding these numbers together, their decimal points must be aligned as in Figure 3.4 (Figure 4-4: Loss of accuracy while aligning decimal points).

Figure 4-4: Loss of accuracy while aligning decimal points

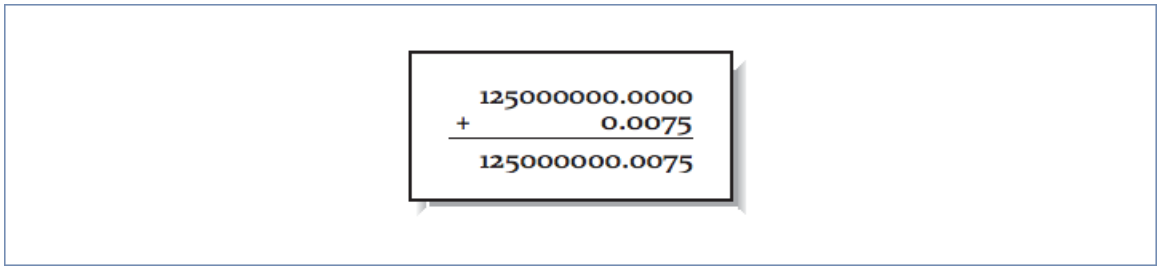


Figure 3.4

Unfortunately, while we have computed the exact result, it cannot fit back into a `REAL*4` variable (7 digits of accuracy) without truncating the 0.0075. So after the addition, the value in Y is exactly 1.25E8. Even sadder, the addition could be performed *millions* of times, and the value for Y would still be 1.25E8.

Because of the limitation on precision, not all algebraic laws apply all the time. For instance, the answer you obtain from X+Y will be the same as Y+X, as per the commutative law for addition. Whichever operand you pick first, the operation yields the same result; they are mathematically equivalent. It also means that you can choose either of the following two forms and get the same answer:

```
(X + Y) + Z
(Y + X) + Z
```

However, this is not equivalent:

```
(Y + Z) + X
```

The third version isn't equivalent to the first two because the order of the calculations has changed. Again, the rearrangement is equivalent algebraically, but not computationally. By changing the order of the calculations, we have taken advantage of the associativity of the operations; we have made an *associative transformation* of the original code.

To understand why the order of the calculations matters, imagine that your computer can perform arithmetic significant to only five decimal places.

Also assume that the values of X, Y, and Z are .00005, .00005, and 1.0000, respectively. This means that:

```
(X + Y) + Z = .00005 + .00005 + 1.0000
            = .0001           + 1.0000     = 1.0001
```

but:

```
(Y + Z) + X =  .00005 + 1.0000 + .00005
            = 1.0000           + .00005   = 1.0000
```

The two versions give slightly different answers. When adding `Y+Z+X`, the sum of the smaller numbers was insignificant when added to the larger number. But when computing `X+Y+Z`, we add the two small numbers first, and their combined sum is large enough to influence the final answer. For this reason, compilers that rearrange operations for the sake of performance generally only do so after the user has requested optimizations beyond the defaults.

For these reasons, the FORTRAN language is very strict about the exact order of evaluation of expressions. To be compliant, the compiler must ensure that the operations occur exactly as you express them.[10]

For Kernighan and Ritchie C, the operator precedence rules are different. Although the precedences between operators are honored (i.e., * comes before +, and evaluation generally occurs left to right for operators of equal precedence), the compiler is allowed to treat a few commutative operations (+, *, &, ^ and |) as if they were fully associative, *even* if they are parenthesized. For instance, you might tell the C compiler:

```
a = x + (y + z);
```

However, the C compiler is free to ignore you, and combine X, Y, and Z in any order it pleases.

Now armed with this knowledge, view the following harmless-looking code segment:

```
REAL*4 SUM,A(1000000)
SUM = 0.0
DO I=1,1000000
  SUM = SUM + A(I)
ENDDO
```

Begins to look like a nightmare waiting to happen. The accuracy of this sum depends of the relative magnitudes and order of the values in the array A. If we sort the array from smallest to largest and then perform the additions, we have a more accurate value. There are other algorithms for computing the sum of an array that reduce the error without requiring a full sort of the data. Consult a good textbook on numerical analysis for the details on these algorithms.

If the range of magnitudes of the values in the array is relatively small, the straight- forward computation of the sum is probably sufficient.

[10] Often even if you didn't mean it.

3.6 Improving Accuracy Using Guard Digits[11]

3.6.1 Improving Accuracy Using Guard Digits

In this section we explore a technique to improve the precision of floating-point computations without using additional storage space for the floating-point numbers.

Consider the following example of a base-10 system with five digits of accuracy performing the following subtraction:

```
10.001 - 9.9993 = 0.0017
```

All of these values can be perfectly represented using our floating-point format. However, if we only have five digits of precision available while aligning the decimal points during the computation, the results end up with significant error as shown in Figure 3.5 (Figure 4-5: Need for guard digits).

Figure 4-5: Need for guard digits

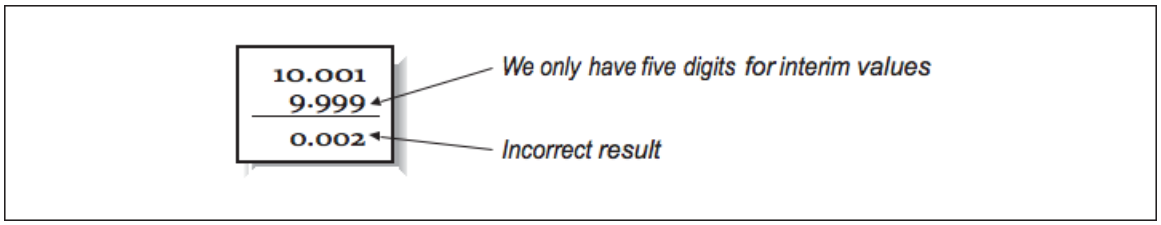


Figure 3.5

To perform this computation and round it correctly, we do not need to increase the number of significant digits for *stored* values. We do, however, need additional digits of precision while performing the computation.

The solution is to add extra *guard digits* which are maintained during the interim steps of the computation. In our case, if we maintained six digits of accuracy while aligning operands, and rounded before normalizing and assigning the final value, we would get the proper result. The guard digits only need to be present as part of the floating-point execution unit in the CPU. It is not necessary to add guard digits to the registers or to the values stored in memory.

It is not necessary to have an extremely large number of guard digits. At some point, the difference in the magnitude between the operands becomes so great that lost digits do not affect the addition or rounding results.

3.7 History of IEEE Floating-Point Format[12]

3.7.1 History of IEEE Floating-Point Format

Prior to the RISC microprocessor revolution, each vendor had their own floating- point formats based on their designers' views of the relative importance of range versus accuracy and speed versus accuracy. It was not uncommon for one vendor to carefully analyze the limitations of another vendor's floating-point format and use this information to convince users that theirs was the only "accurate" floating- point implementation. In reality none of the formats was perfect. The formats were simply imperfect in different ways.

[11] This content is available online at <http://cnx.org/content/m32744/1.1/>.
[12] This content is available online at <http://cnx.org/content/m32770/1.1/>.

During the 1980s the Institute for Electrical and Electronics Engineers (IEEE) produced a standard for the floating-point format. The title of the standard is "IEEE 754-1985 Standard for Binary Floating-Point Arithmetic." This standard provided the precise definition of a floating-point format and described the operations on floating-point values.

Because IEEE 754 was developed after a variety of floating-point formats had been in use for quite some time, the IEEE 754 working group had the benefit of examining the existing floating-point designs and taking the strong points, and avoiding the mistakes in existing designs. The IEEE 754 specification had its beginnings in the design of the Intel i8087 floating-point coprocessor. The i8087 floating-point format improved on the DEC VAX floating-point format by adding a number of significant features.

The near universal adoption of IEEE 754 floating-point format has occurred over a 10-year time period. The high performance computing vendors of the mid 1980s (Cray IBM, DEC, and Control Data) had their own proprietary floating-point formats that they had to continue supporting because of their installed user base. They really had no choice but to continue to support their existing formats. In the mid to late 1980s the primary systems that supported the IEEE format were RISC workstations and some coprocessors for microprocessors. Because the designers of these systems had no need to protect a proprietary floating-point format, they readily adopted the IEEE format. As RISC processors moved from general-purpose integer computing to high performance floating-point computing, the CPU designers found ways to make IEEE floating-point operations operate very quickly. In 10 years, the IEEE 754 has gone from a standard for floating-point coprocessors to the dominant floating-point standard for all computers. Because of this standard, we, the users, are the beneficiaries of a portable floating-point environment.

3.7.2 IEEE Floating-Point Standard

The IEEE 754 standard specified a number of different details of floating-point operations, including:

- Storage formats
- Precise specifications of the results of operations
- Special values
- Specified runtime behavior on illegal operations

Specifying the floating-point format to this level of detail insures that when a computer system is compliant with the standard, users can expect repeatable execution from one hardware platform to another when operations are executed in the same order.

3.7.3 IEEE Storage Format

The two most common IEEE floating-point formats in use are 32- and 64-bit numbers. Table 3.1: Table 4-1: Parameters of IEEE 32- and 64-Bit Formats gives the general parameters of these data types.

Table 4-1: Parameters of IEEE 32- and 64-Bit Formats

IEEE75	FORTRAN	C	Bits	Exponent Bits	Mantissa Bits
Single	REAL*4	float	32	8	24
Double	REAL*8	double	64	11	53
Double-Extended	REAL*10	long double	>=80	>=15	>=64

Table 3.1

In FORTRAN, the 32-bit format is usually called REAL, and the 64-bit format is usually called DOUBLE. However, some FORTRAN compilers double the sizes for these data types. For that reason, it is safest to declare your FORTRAN variables as `REAL*4` or `REAL*8`. The double-extended format is not as well supported

in compilers and hardware as the single- and double-precision formats. The bit arrangement for the single and double formats are shown in Figure 3.6 (Figure 4-6: IEEE754 floating-point formats).

Based on the storage layouts in Table 3.1: Table 4-1: Parameters of IEEE 32- and 64-Bit Formats, we can derive the ranges and accuracy of these formats, as shown in Table 3.2: Table 4-2: Range and Accuracy of IEEE 32- and 64-Bit Formats.

Figure 4-6: IEEE754 floating-point formats

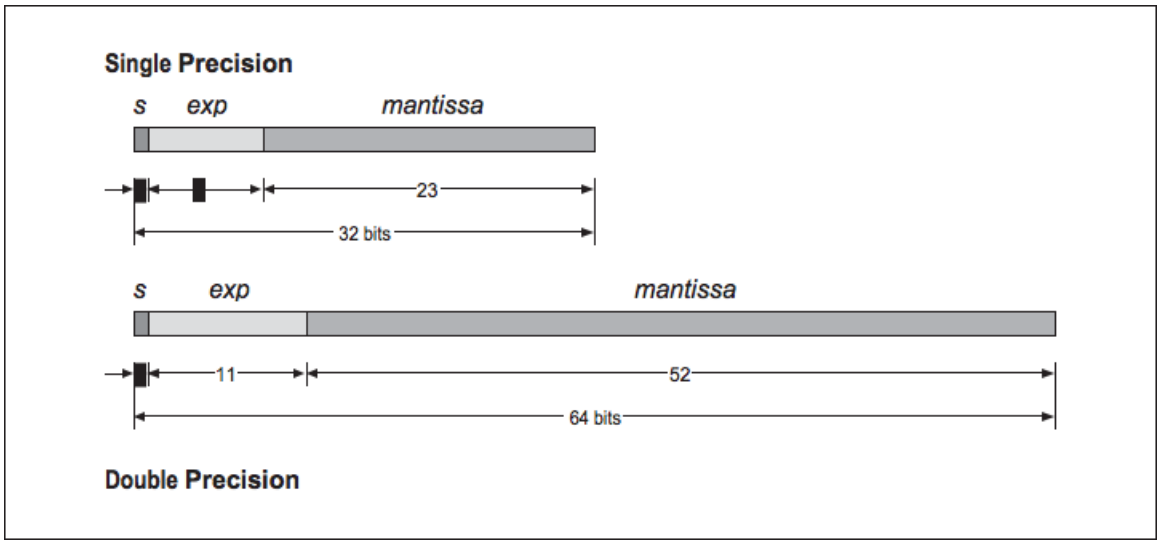


Figure 3.6

Table 4-2: Range and Accuracy of IEEE 32- and 64-Bit Formats

IEEE754	Minimum Normalized Number	Largest Finite Number	Base-10 Accuracy
Single	1.2E-38	3.4 E+38	6-9 digits
Double	2.2E-308	1.8 E+308	15-17 digits
Extended Double	3.4E-4932	1.2 E+4932	18-21 digits

Table 3.2

3.7.3.1 Converting from Base-10 to IEEE Internal Format

We now examine how a 32-bit floating-point number is stored. The high-order bit is the sign of the number. Numbers are stored in a sign-magnitude format (i.e., not 2's - complement). The exponent is stored in the 8-bit field biased by adding 127 to the exponent. This results in an exponent ranging from -126 through +127.

The mantissa is converted into base-2 and normalized so that there is one nonzero digit to the left of the binary place, adjusting the exponent as necessary. The digits to the right of the binary point are then stored in the low-order 23 bits of the word. Because all numbers are normalized, there is no need to store the leading 1.

This gives a free extra bit of precision. Because this bit is dropped, it's no longer proper to refer to the stored value as the mantissa. In IEEE parlance, this mantissa minus its leading digit is called the *significand*.

Figure 3.7 (Figure 4-7: Converting from base-10 to IEEE 32-bit format) shows an example conversion from base-10 to IEEE 32-bit format.

Figure 4-7: Converting from base-10 to IEEE 32-bit format

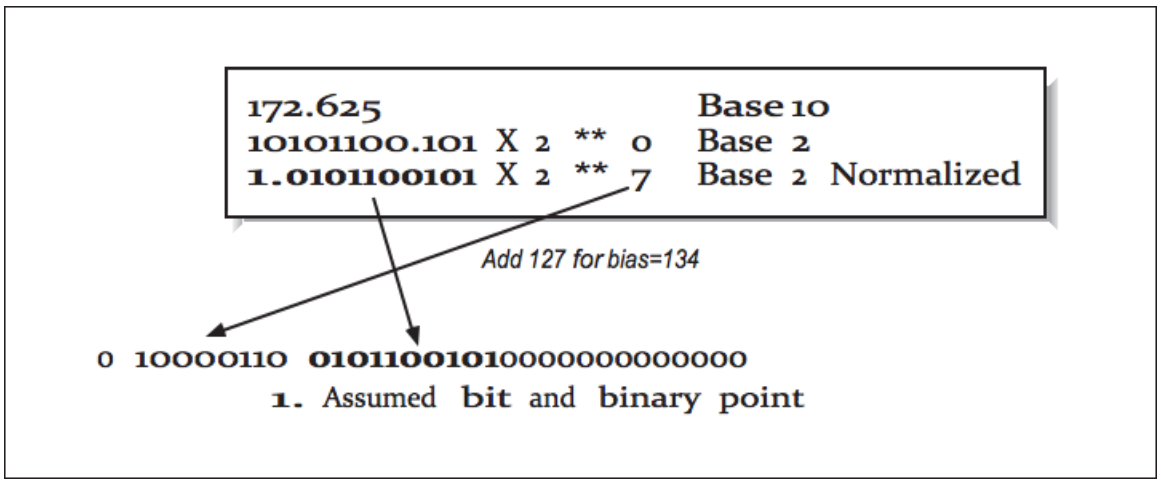


Figure 3.7

The 64-bit format is similar, except the exponent is 11 bits long, biased by adding 1023 to the exponent, and the significand is 54 bits long.

3.8 IEEE Operations[13]

3.8.1 IEEE Operations

The IEEE standard specifies how computations are to be performed on floating- point values on the following operations:

- Addition
- Subtraction
- Multiplication
- Division
- Square root
- Remainder (modulo)
- Conversion to/from integer
- Conversion to/from printed base-10

These operations are specified in a machine-independent manner, giving flexibility to the CPU designers to implement the operations as efficiently as possible while maintaining compliance with the standard. During operations, the IEEE standard requires the maintenance of two guard digits and a sticky bit for intermediate values. The guard digits above and the sticky bit are used to indicate if any of the bits beyond the second guard digit is nonzero.

[13] This content is available online at <http://cnx.org/content/m32756/1.1/>.

Figure 4-8: Computation using guard and sticky bits

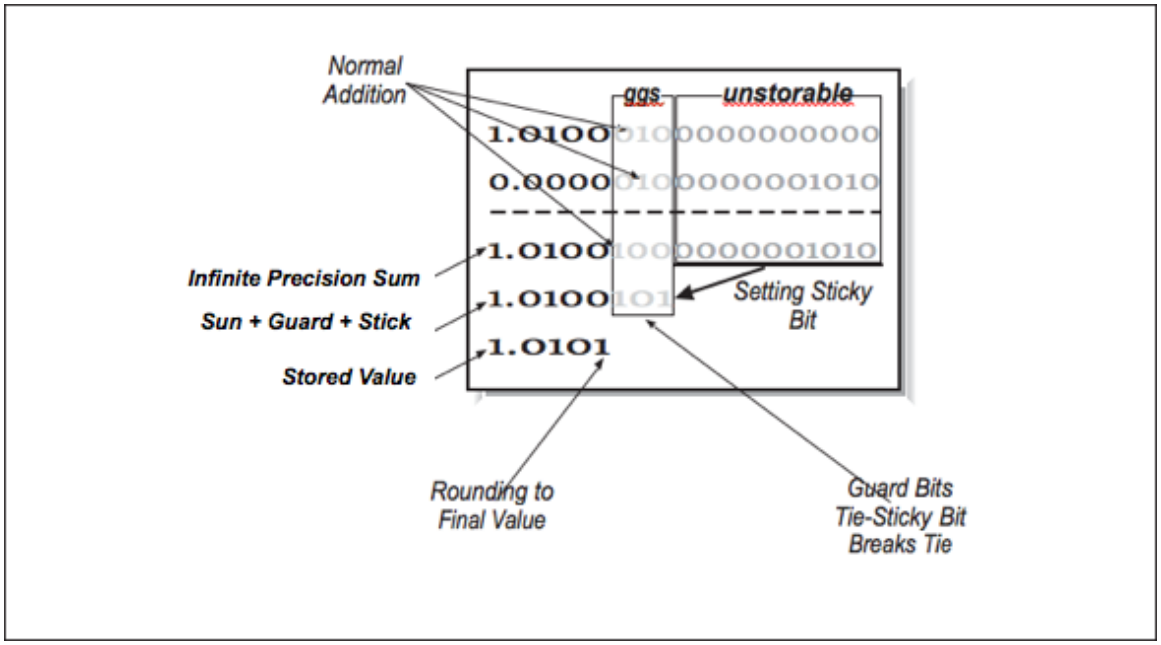


Figure 3.8

In Figure 3.8 (Figure 4-8: Computation using guard and sticky bits), we have five bits of normal precision, two guard digits, and a sticky bit. Guard bits simply operate as normal bits — as if the significand were 25 bits. Guard bits participate in rounding as the extended operands are added. The sticky bit is set to 1 if any of the bits beyond the guard bits is nonzero in either operand.[14] Once the extended sum is computed, it is rounded so that the value stored in memory is the closest possible value to the extended sum including the guard digits. Table 3.3: Table 4-3: Extended Sums and Their Stored Values shows all eight possible values of the two guard digits and the sticky bit and the resulting stored value with an explanation as to why.

[14] If you are somewhat hardware-inclined and you think about it for a moment, you will soon come up with a way to properly maintain the sticky bit without ever computing the full "infinite precision sum." You just have to keep track as things get shifted around.

Table 4-3: Extended Sums and Their Stored Values

Extended Sum	Stored Value	Why
1.0100 000	1.0100	Truncated based on guard digits
1.0100 001	1.0100	Truncated based on guard digits
1.0100 010	1.0100	Rounded down based on guard digits
1.0100 011	1.0100	Rounded down based on guard digits
1.0100 100	1.0100	Rounded down based on sticky bit
1.0100 101	1.0101	Rounded up based on sticky bit
1.0100 110	1.0101	Rounded up based on guard digits
1.0100 111	1.0101	Rounded up based on guard digits

Table 3.3

The first priority is to check the guard digits. Never forget that the sticky bit is just a hint, not a real digit. So if we can make a decision without looking at the sticky bit, that is good. The only decision we are making is to round the last storable bit up or down. When that stored value is retrieved for the next computation, its guard digits are set to zeros. It is sometimes helpful to think of the stored value as having the guard digits, but set to zero.

Two guard digits and the sticky bit in the IEEE format insures that operations yield the same rounding as if the intermediate result were computed using unlimited precision and then rounded to fit within the limits of precision of the final computed value.

At this point, you might be asking, "Why do I care about this minutiae?" At some level, unless you are a hardware designer, you don't care. But when you examine details like this, you can be assured of one thing: when they developed the IEEE floating-point standard, they looked at the details *very* carefully. The goal was to produce the most accurate possible floating-point standard within the constraints of a fixed-length 32- or 64-bit format. Because they did such a good job, it's one less thing you have to worry about. Besides, this stuff makes great exam questions.

3.9 Special Values[15]

3.9.1 Special Values

In addition to specifying the results of operations on numeric data, the IEEE standard also specifies the precise behavior on undefined operations such as dividing by zero. These results are indicated using several special values. These values are bit patterns that are stored in variables that are checked before operations are performed. The IEEE operations are all defined on these special values in addition to the normal numeric values. Table 3.4: Table 4-4: Special Values for an IEEE 32-Bit Number summarizes the special values for a 32-bit IEEE floating-point number.

[15]This content is available online at <http://cnx.org/content/m32758/1.1/>.

Table 4-4: Special Values for an IEEE 32-Bit Number

Special Value	Exponent	Significand
+ or − 0	00000000	0
Denormalized number	00000000	nonzero
NaN (Not a Number)	11111111	nonzero
+ or − Infinity	11111111	0

Table 3.4

The value of the exponent and significand determines which type of special value this particular floating-point number represents. Zero is designed such that integer zero and floating-point zero are the same bit pattern.

Denormalized numbers can occur at some point as a number continues to get smaller, and the exponent has reached the minimum value. We could declare that minimum to be the smallest representable value. However, with denormalized values, we can continue by setting the exponent bits to zero and shifting the significand bits to the right, first adding the leading "1" that was dropped, then continuing to add leading zeros to indicate even smaller values. At some point the last nonzero digit is shifted off to the right, and the value becomes zero. This approach is called *gradual underflow* where the value keeps approaching zero and then eventually becomes zero. Not all implementations support denormalized numbers in hardware; they might trap to a software routine to handle these numbers at a significant performance cost.

At the top end of the biased exponent value, an exponent of all 1s can represent the *Not a Number* (NaN) value or infinity. Infinity occurs in computations roughly according to the principles of mathematics. If you continue to increase the magnitude of a number beyond the range of the floating-point format, once the range has been exceeded, the value becomes infinity. Once a value is infinity, further additions won't increase it, and subtractions won't decrease it. You can also produce the value infinity by dividing a nonzero value by zero. If you divide a nonzero value by infinity, you get zero as a result.

The NaN value indicates a number that is not mathematically defined. You can generate a NaN by dividing zero by zero, dividing infinity by infinity, or taking the square root of -1. The difference between infinity and NaN is that the NaN value has a nonzero significand. The NaN value is very sticky. Any operation that has a NaN as one of its inputs always produces a NaN result.

3.10 Exceptions and Traps[16]

3.10.1 Exceptions and Traps

In addition to defining the results of computations that aren't mathematically defined, the IEEE standard provides programmers with the ability to detect when these special values are being produced. This way, programmers can write their code without adding extensive IF tests throughout the code checking for the magnitude of values. Instead they can register a trap handler for an event such as underflow and handle the event when it occurs. The exceptions defined by the IEEE standard include:

- Overflow to infinity
- Underflow to zero
- Division by zero
- Invalid operation
- Inexact operation

[16] This content is available online at <http://cnx.org/content/m32760/1.1/>.

According to the standard, these traps are under the control of the user. In most cases, the compiler runtime library manages these traps under the direction from the user through compiler flags or runtime library calls. Traps generally have significant overhead compared to a single floating-point instruction, and if a program is continually executing trap code, it can significantly impact performance.

In some cases it's appropriate to ignore traps on certain operations. A commonly ignored trap is the underflow trap. In many iterative programs, it's quite natural for a value to keep reducing to the point where it "disappears." Depending on the application, this may or may not be an error situation so this exception can be safely ignored.

If you run a program and then it terminates, you see a message such as:

```
Overflow handler called 10,000,000 times
```

It probably means that you need to figure out why your code is exceeding the range of the floating-point format. It probably also means that your code is executing more slowly because it is spending too much time in its error handlers.

3.11 Compiler Issues[17]

3.11.1 Compiler Issues

The IEEE 754 floating-point standard does a good job describing how floating- point operations are to be performed. However, we generally don't write assembly language programs. When we write in a higher-level language such as FORTRAN, it's sometimes difficult to get the compiler to generate the assembly language you need for your application. The problems fall into two categories:

- The compiler is too conservative in trying to generate IEEE-compliant code and produces code that doesn't operate at the peak speed of the processor. On some processors, to fully support gradual underflow, extra instructions must be generated for certain instructions. If your code will never underflow, these instructions are unnecessary overhead.
- The optimizer takes liberties rewriting your code to improve its performance, eliminating some necessary steps. For example, if you have the following code:

  ```
  Z = X + 500
  Y = Z - 200
  ```

  The optimizer may replace it with `Y = X + 300`. However, in the case of a value for `X` that is close to overflow, the two sequences may not produce the same result.

Sometimes a user prefers "fast" code that loosely conforms to the IEEE standard, and at other times the user will be writing a numerical library routine and need total control over each floating-point operation. Compilers have a challenge supporting the needs of both of these types of users. Because of the nature of the high performance computing market and benchmarks, often the "fast and loose" approach prevails in many compilers.

[17] This content is available online at <http://cnx.org/content/m32762/1.1/>.

3.12 Closing Notes[18]

3.12.1 Closing Notes

While this is a relatively long chapter with a lot of technical detail, it does not even begin to scratch the surface of the IEEE floating-point format or the entire field of numerical analysis. We as programmers must be careful about the accuracy of our programs, lest the results become meaningless. Here are a few basic rules to get you started:

- Look for compiler options that relax or enforce strict IEEE compliance and choose the appropriate option for your program. You may even want to change these options for different portions of your program.
- Use `REAL*8` for computations unless you are sure `REAL*4` has sufficient precision. Given that REAL*4 has roughly 7 digits of precision, if the bottom digits become meaningless due to rounding and computations, you are in some danger of seeing the effect of the errors in your results. `REAL*8` with 13 digits makes this much less likely to happen.
- Be aware of the relative magnitude of numbers when you are performing additions.
- When summing up numbers, if there is a wide range, sum from smallest to largest.
- Perform multiplications before divisions whenever possible.
- When performing a comparison with a computed value, check to see if the values are "close" rather than identical.
- Make sure that you are not performing any unnecessary type conversions during the critical portions of your code.

An excellent reference on floating-point issues and the IEEE format is "What Every Computer Scientist Should Know About Floating-Point Arithmetic," written by David Goldberg, in ACM Computing Surveys magazine (March 1991). This article gives examples of the most common problems with floating-point and outlines the solutions. It also covers the IEEE floating-point format very thoroughly. I also recommend you consult Dr. William Kahan's home page (http://www.cs.berkeley.edu/~wkahan/[19]) for some excellent materials on the IEEE format and challenges using floating-point arithmetic. Dr. Kahan was one of the original designers of the Intel i8087 and the IEEE 754 floating-point format.

3.13 Exercises[20]

3.13.1 Exercises

Exercise 3.1
Run the following code to count the number of inverses that are not perfectly accurate:

```
REAL*4 X,Y,Z
INTEGER I
I = 0
DO X=1.0,1000.0,1.0
  Y = 1.0 / X
  Z = Y * X
  IF ( Z .NE. 1.0 ) THEN
```

[18] This content is available online at <http://cnx.org/content/m32768/1.1/>.
[19] http://www.cs.berkeley.edu/~wkahan/
[20] This content is available online at <http://cnx.org/content/m32765/1.1/>.

```
      I = I + 1
    ENDIF
  ENDDO
  PRINT *,'Found ',I
  END
```

Exercise 3.2
Change the type of the variables to `REAL*8` and repeat. Make sure to keep the optimization at a sufficiently low level (-00) to keep the compiler from eliminating the computations.

Exercise 3.3
Write a program to determine the number of digits of precision for `REAL*4` and `REAL*8`.

Exercise 3.4
Write a program to demonstrate how summing an array forward to backward and backward to forward can yield a different result.

Exercise 3.5
Assuming your compiler supports varying levels of IEEE compliance, take a significant computational code and test its overall performance under the various IEEE compliance options. Do the results of the program change?

Chapter 4

Understanding Parallelism

4.1 Introduction[1]

4.1.1 Understanding Parallelism

In a sense, we have been talking about parallelism from the beginning of the book. Instead of calling it "parallelism," we have been using words like "pipelined," "superscalar," and "compiler flexibility." As we move into programming on multiprocessors, we must increase our understanding of parallelism in order to understand how to effectively program these systems. In short, as we gain more parallel resources, we need to find more parallelism in our code.

When we talk of parallelism, we need to understand the concept of granularity. The granularity of parallelism indicates the size of the computations that are being performed at the same time between synchronizations. Some examples of parallelism in order of increasing grain size are:

- When performing a 32-bit integer addition, using a carry lookahead adder, you can partially add bits 0 and 1 at the same time as bits 2 and 3.
- On a pipelined processor, while decoding one instruction, you can fetch the next instruction.
- On a two-way superscalar processor, you can execute any combination of an integer and a floating-point instruction in a single cycle.
- On a multiprocessor, you can divide the iterations of a loop among the four processors of the system.
- You can split a large array across four workstations attached to a network. Each workstation can operate on its local information and then exchange boundary values at the end of each time step.

In this chapter, we start at *instruction-level parallelism* (pipelined and superscalar) and move toward *thread-level parallelism*, which is what we need for multiprocessor systems. It is important to note that the different levels of parallelism are generally not in conflict. Increasing thread parallelism at a coarser grain size often exposes more fine-grained parallelism.

The following is a loop that has plenty of parallelism:

```
DO I=1,16000
  A(I) = B(I) * 3.14159
ENDDO
```

[1] This content is available online at <http://cnx.org/content/m32775/1.1/>.

We have expressed the loop in a way that would imply that A(1) must be computed first, followed by A(2), and so on. However, once the loop was completed, it would not have mattered if A(16000), were computed first followed by A(15999), and so on. The loop could have computed the even values of I and then computed the odd values of I. It would not even make a difference if all 16,000 of the iterations were computed simultaneously using a 16,000-way superscalar processor.[2] If the compiler has flexibility in the order in which it can execute the instructions that make up your program, it can execute those instructions simultaneously when parallel hardware is available.

One technique that computer scientists use to formally analyze the potential parallelism in an algorithm is to characterize how quickly it would execute with an "infinite-way" superscalar processor.

Not all loops contain as much parallelism as this simple loop. We need to identify the things that limit the parallelism in our codes and remove them whenever possible. In previous chapters we have already looked at removing clutter and rewriting loops to simplify the body of the loop.

This chapter also supplements Chapter 5, *What a Compiler Does*, in many ways. We looked at the mechanics of compiling code, all of which apply here, but we didn't answer all of the "whys." Basic block analysis techniques form the basis for the work the compiler does when looking for more parallelism. Looking at two pieces of data, instructions, or data and instructions, a modern compiler asks the question, "Do these things depend on each other?" The three possible answers are yes, no, and we don't know. The third answer is effectively the same as a yes, because a compiler has to be conservative whenever it is unsure whether it is safe to tweak the ordering of instructions.

Helping the compiler recognize parallelism is one of the basic approaches specialists take in tuning code. A slight rewording of a loop or some supplementary information supplied to the compiler can change a "we don't know" answer into an opportunity for parallelism. To be certain, there are other facets to tuning as well, such as optimizing memory access patterns so that they best suit the hardware, or recasting an algorithm. And there is no single best approach to every problem; any tuning effort has to be a combination of techniques.

4.2 Dependencies[3]

4.2.1 Dependencies

Imagine a symphony orchestra where each musician plays without regard to the conductor or the other musicians. At the first tap of the conductor's baton, each musician goes through all of his or her sheet music. Some finish far ahead of others, leave the stage, and go home. The cacophony wouldn't resemble music (come to think of it, it would resemble experimental jazz) because it would be totally uncoordinated. Of course this isn't how music is played. A computer program, like a musical piece, is woven on a fabric that unfolds in time (though perhaps woven more loosely). Certain things must happen before or along with others, and there is a rate to the whole process.

With computer programs, whenever event A must occur before event B can, we say that B is *dependent* on A. We call the relationship between them a dependency. Sometimes dependencies exist because of calculations or memory operations; we call these *data dependencies*. Other times, we are waiting for a branch or do-loop exit to take place; this is called a *control dependency*. Each is present in every program to varying degrees. The goal is to eliminate as many dependencies as possible. Rearranging a program so that two chunks of the computation are less dependent exposes *parallelism*, or opportunities to do several things at once.

[2] Interestingly, this is not as far-fetched as it might seem. On a single instruction multiple data (SIMD) computer such as the Connection CM-2 with 16,384 processors, it would take three instruction cycles to process this entire loop. See Chapter 12, Large-Scale Parallel Computing, for more details on this type of architecture.

[3] This content is available online at <http://cnx.org/content/m32777/1.1/>.

4.2.1.1 Control Dependencies

Just as variable assignments can depend on other assignments, a variable's value can also depend on the *flow of control* within the program. For instance, an assignment within an if-statement can occur only if the conditional evaluates to true. The same can be said of an assignment within a loop. If the loop is never entered, no statements inside the loop are executed.

When calculations occur as a consequence of the flow of control, we say there is a *control dependency*, as in the code below and shown graphically in Figure 4.1 (Figure 9-1: Control dependency). The assignment located inside the block-if may or may not be executed, depending on the outcome of the test `X .NE. 0.` In other words, the value of `Y` depends on the flow of control in the code around it. Again, this may sound to you like a concern for compiler designers, not programmers, and that's mostly true. But there are times when you might want to move control-dependent instructions around to get expensive calculations out of the way (provided your compiler isn't smart enough to do it for you). For example, say that Figure 4.2 (Figure 9-2: A little section of your program) represents a little section of your program. Flow of control enters at the top and goes through two branch decisions. Furthermore, say that there is a square root operation at the entry point, and that the flow of control almost always goes from the top, down to the leg containing the statement `A=0.0`. This means that the results of the calculation `A=SQRT(B)` are almost always discarded because A gets a new value of 0.0 each time through. A square root operation is always "expensive" because it takes a lot of time to execute. The trouble is that you can't just get rid of it; occasionally it's needed. However, you could move it out of the way and continue to observe the control dependencies by making two copies of the square root operation along the less traveled branches, as shown in Figure 4.3 (Figure 9-3: Expensive operation moved so that it's rarely executed). This way the `SQRT` would execute only along those paths where it was actually needed.

Figure 9-1: Control dependency

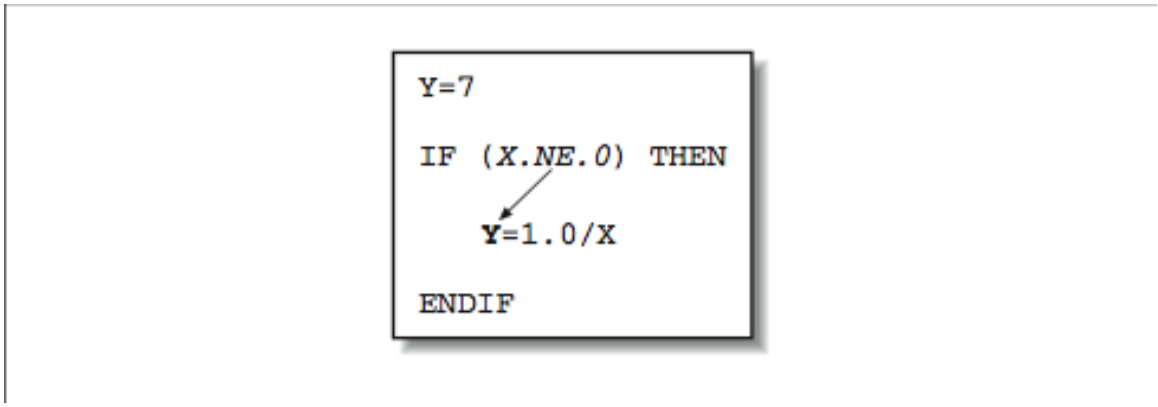


Figure 4.1

Figure 9-2: A little section of your program

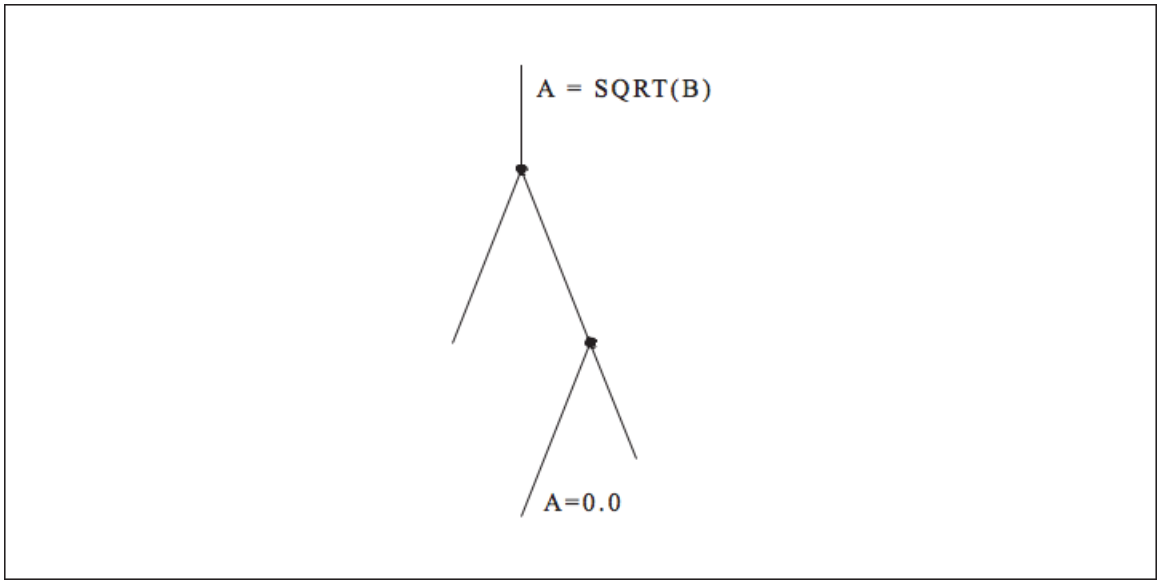


Figure 4.2

This kind of instruction scheduling will be appearing in compilers (and even hardware) more and more as time goes on. A variation on this technique is to calculate results that might be needed at times when there is a gap in the instruction stream (because of dependencies), thus using some spare cycles that might otherwise be wasted.

Figure 9-3: Expensive operation moved so that it's rarely executed

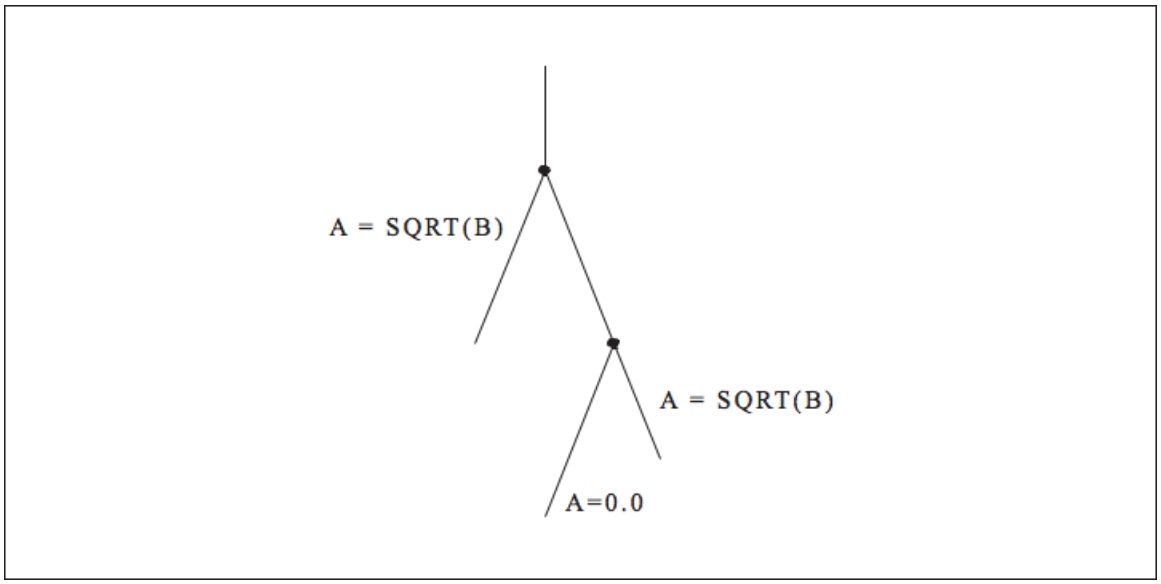


Figure 4.3

4.2.1.2 Data Dependencies

A calculation that is in some way bound to a previous calculation is said to be data dependent upon that calculation. In the code below, the value of `B` is data dependent on the value of `A`. That's because you can't calculate `B` until the value of `A` is available:

```
A = X + Y + COS(Z)
B = A * C
```

This dependency is easy to recognize, but others are not so simple. At other times, you must be careful not to rewrite a variable with a new value before every other computation has finished using the old value. We can group all data dependencies into three categories: (1) flow dependencies, (2) antidependencies, and (3) output dependencies. Figure 4.4 (Figure 9-4: Types of data dependencies) contains some simple examples to demonstrate each type of dependency. In each example, we use an arrow that starts at the source of the dependency and ends at the statement that must be delayed by the dependency. The key problem in each of these dependencies is that the second statement can't execute until the first has completed. Obviously in the particular output dependency example, the first computation is dead code and can be eliminated unless there is some intervening code that needs the values. There are other techniques to eliminate either output or antidependencies. The following example contains a flow dependency followed by an output dependency.

Figure 9-4: Types of data dependencies

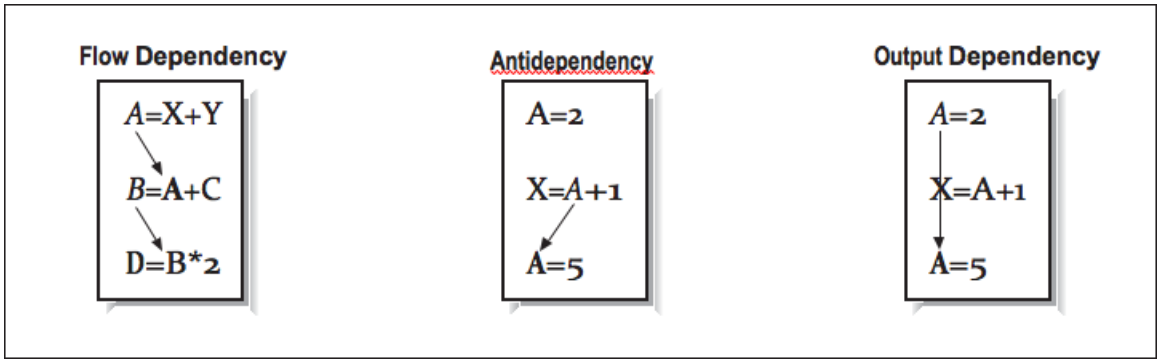


Figure 4.4

```
X = A / B
Y = X + 2.0
X = D - E
```

While we can't eliminate the flow dependency, the output dependency can be eliminated by using a scratch variable:

```
Xtemp = A/B
Y = Xtemp + 2.0
X = D - E
```

As the number of statements and the interactions between those statements increase, we need a better way to identify and process these dependencies. Figure 4.5 (Figure 9-5: Multiple dependencies) shows four statements with four dependencies.

Figure 9-5: Multiple dependencies

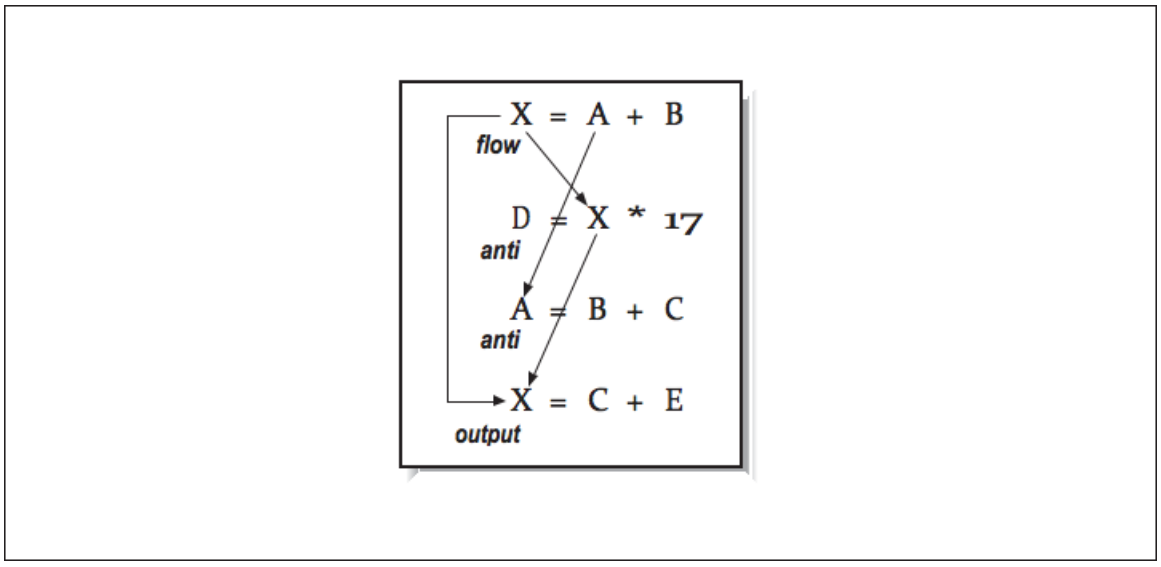


Figure 4.5

None of the second through fourth instructions can be started before the first instruction completes.

4.2.1.3 Forming a DAG

One method for analyzing a sequence of instructions is to organize it into a *directed acyclic graph* (DAG).[4] Like the instructions it represents, a DAG describes all of the calculations and relationships between variables. The data flow within a DAG proceeds in one direction; most often a DAG is constructed from top to bottom. Identifiers and constants are placed at the "leaf " nodes — the ones on the top. Operations, possibly with variable names attached, make up the internal nodes. Variables appear in their final states at the bottom. The DAG's edges order the relationships between the variables and operations within it. All data flow proceeds from top to bottom.

To construct a DAG, the compiler takes each intermediate language tuple and maps it onto one or more nodes. For instance, those tuples that represent binary operations, such as addition (`X=A+B`), form a portion of the DAG with two inputs (`A` and `B`) bound together by an operation (`+`). The result of the operation may feed into yet other operations within the basic block (and the DAG) as shown in Figure 4.6 (Figure 9-6: A trivial data flow graph).

[4] A graph is a collection of nodes connected by edges. By directed, we mean that the edges can only be traversed in specified directions. The word acyclic means that there are no cycles in the graph; that is, you can't loop anywhere within it.

Figure 9-6: A trivial data flow graph

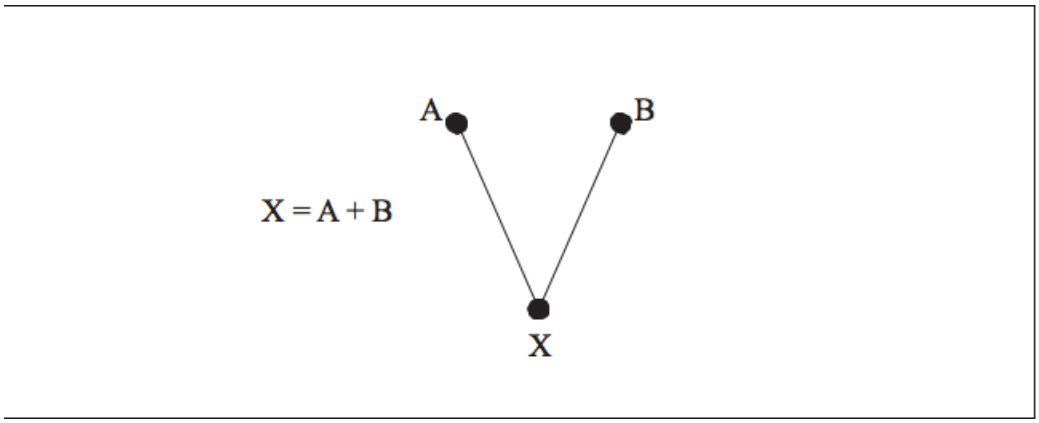


Figure 4.6

For a basic block of code, we build our DAG in the order of the instructions. The DAG for the previous four instructions is shown in Figure 4.7 (Figure 9-7: A more complex data flow graph). This particular example has many dependencies, so there is not much opportunity for parallelism. Figure 4.8 (Figure 9-8: Extracting parallelism from a DAG) shows a more straightforward example shows how constructing a DAG can identify parallelism.

From this DAG, we can determine that instructions 1 and 2 can be executed in parallel. Because we see the computations that operate on the values `A` and `B` while processing instruction 4, we can eliminate a common subexpression during the construction of the DAG. If we can determine that `Z` is the only variable that is used outside this small block of code, we can assume the `Y` computation is dead code.

Figure 9-7: A more complex data flow graph

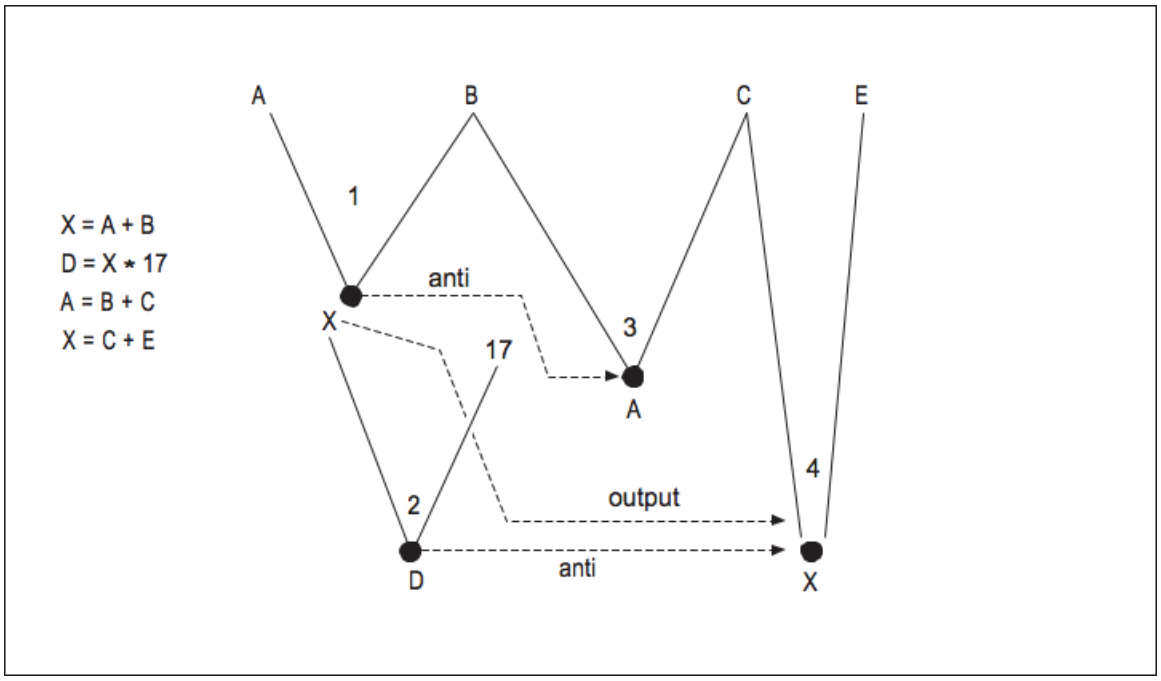


Figure 4.7

By constructing the DAG, we take a sequence of instructions and determine which must be executed in a particular order and which can be executed in parallel. This type of data flow analysis is very important in the codegeneration phase on super-scalar processors. We have introduced the concept of dependencies and how to use data flow to find opportunities for parallelism in code sequences within a basic block. We can also use data flow analysis to identify dependencies, opportunities for parallelism, and dead code between basic blocks.

4.2.1.4 Uses and Definitions

As the DAG is constructed, the compiler can make lists of variable uses and *definitions*, as well as other information, and apply these to global optimizations across many basic blocks taken together. Looking at the DAG in Figure 4.8 (Figure 9-8: Extracting parallelism from a DAG), we can see that the variables defined are `Z`, `Y`, `X`, `C`, and `D`, and the variables used are `A` and `B`. Considering many basic blocks at once, we can say how far a particular variable definition reaches — where its value can be seen. From this we can recognize situations where calculations are being discarded, where two uses of a given variable are completely independent, or where we can overwrite register-resident values without saving them back to memory. We call this investigation *data flow analysis*.

Figure 9-8: Extracting parallelism from a DAG

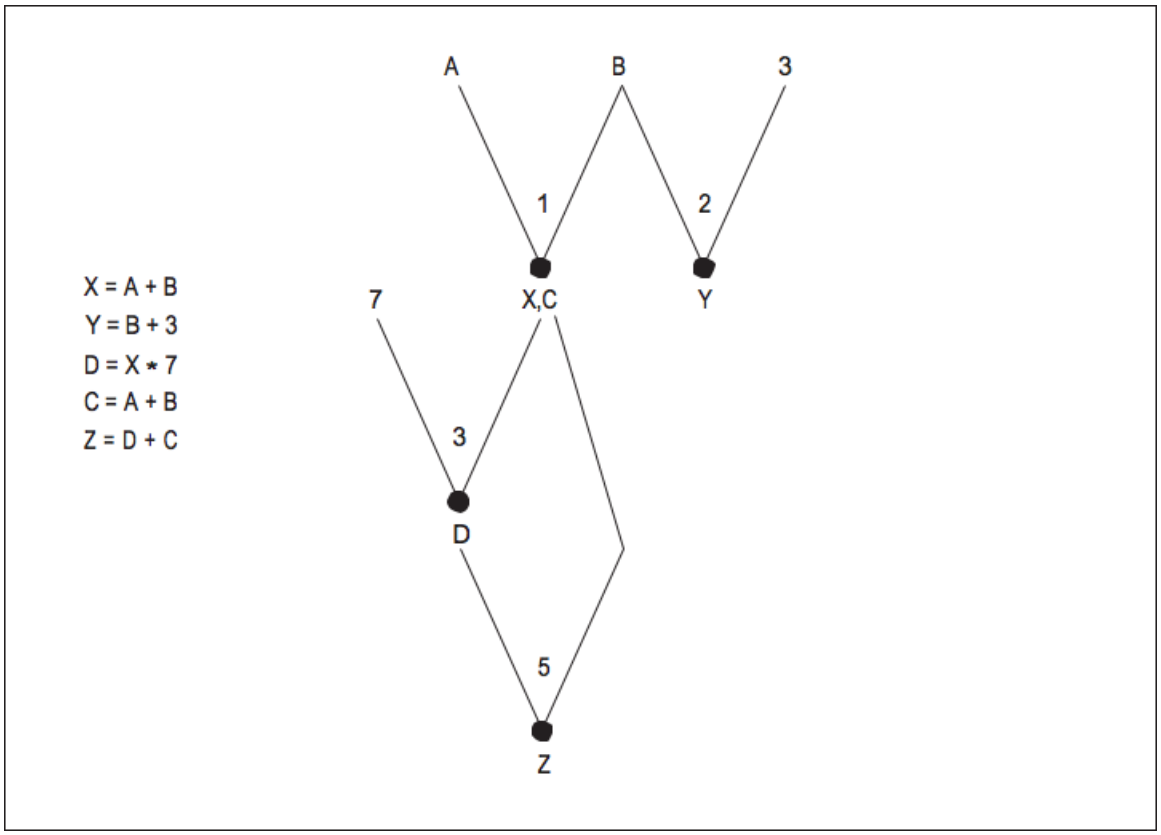


Figure 4.8

To illustrate, suppose that we have the flow graph in Figure 4.9 (Figure 9-9: Flow graph for data flow analysis). Beside each basic block we've listed the variables it uses and the variables it defines. What can data flow analysis tell us?

Notice that a value for `A` is defined in block `X` but only used in block `Y`. That means that `A` is dead upon exit from block `Y` or immediately upon taking the right-hand branch leaving `X`; none of the other basic blocks uses the value of A. That tells us that any associated resources, such as a register, can be freed for other uses.

Looking at Figure 4.9 (Figure 9-9: Flow graph for data flow analysis) we can see that `D` is defined in basic block `X`, but never used. This means that the calculations defining `D` can be discarded.

Something interesting is happening with the variable `G`. Blocks `X` and `W` both use it, but if you look closely you'll see that the two uses are distinct from one another, meaning that they can be treated as two independent variables.

A compiler featuring advanced instruction scheduling techniques might notice that `W` is the only block that uses the value for `E`, and so move the calculations defining `E` out of block `Y` and into `W`, where they are needed.

Figure 9-9: Flow graph for data flow analysis

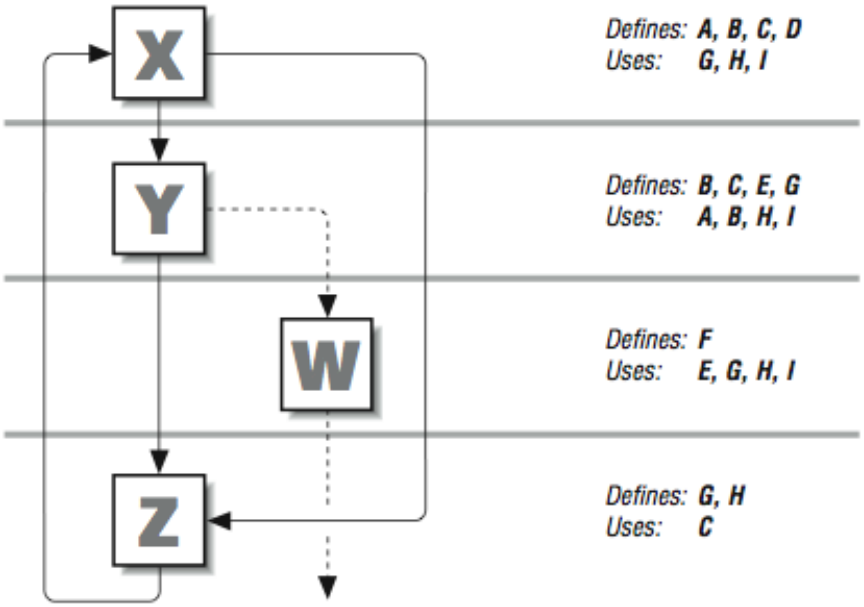


Figure 4.9

In addition to gathering data about variables, the compiler can also keep information about subexpressions. Examining both together, it can recognize cases where redundant calculations are being made (across basic blocks), and substitute previously computed values in their place. If, for instance, the expression `H*I` appears in blocks `X`, `Y`, and `W`, it could be calculated just once in block `X` and propagated to the others that use it.

4.3 Loops[5]

4.3.1 Loops

Loops are the center of activity for many applications, so there is often a high payback for simplifying or moving calculations outside, into the computational suburbs. Early compilers for parallel architectures used pattern matching to identify the bounds of their loops. This limitation meant that a hand-constructed loop using if-statements and goto-statements would not be correctly identified as a loop. Because modern compilers use data flow graphs, it's practical to identify loops as a particular subset of nodes in the flow graph. To a data flow graph, a hand constructed loop looks the same as a compiler-generated loop. Optimizations can therefore be applied to either type of loop.

[5]This content is available online at <http://cnx.org/content/m32784/1.1/>.

Once we have identified the loops, we can apply the same kinds of data-flow analysis we applied above. Among the things we are looking for are calculations that are unchanging within the loop and variables that change in a predictable (linear) fashion from iteration to iteration.

How does the compiler identify a loop in the flow graph? Fundamentally, two conditions have to be met:

- A given node has to dominate all other nodes within the suspected loop. This means that all paths to any node in the loop have to pass through one particular node, the dominator. The dominator node forms the header at the top of the loop.
- There has to be a cycle in the graph. Given a dominator, if we can find a path back to it from one of the nodes it dominates, we have a loop. This path back is known as the *back edge* of the loop.

The flow graph in Figure 4.10 (Figure 9-10: Flowgraph with a loop in it) contains one loop and one red herring. You can see that node B dominates every node below it in the subset of the flow graph. That satisfies Condition 1 and makes it a candidate for a loop header. There is a path from `E` to `B`, and `B` dominates `E`, so that makes it a back edge, satisfying Condition 2. Therefore, the nodes `B`, `C`, `D`, and `E` form a loop. The loop goes through an array of linked list start pointers and traverses the lists to determine the total number of nodes in all lists. Letters to the extreme right correspond to the basic block numbers in the flow graph.

Figure 9-10: Flowgraph with a loop in it

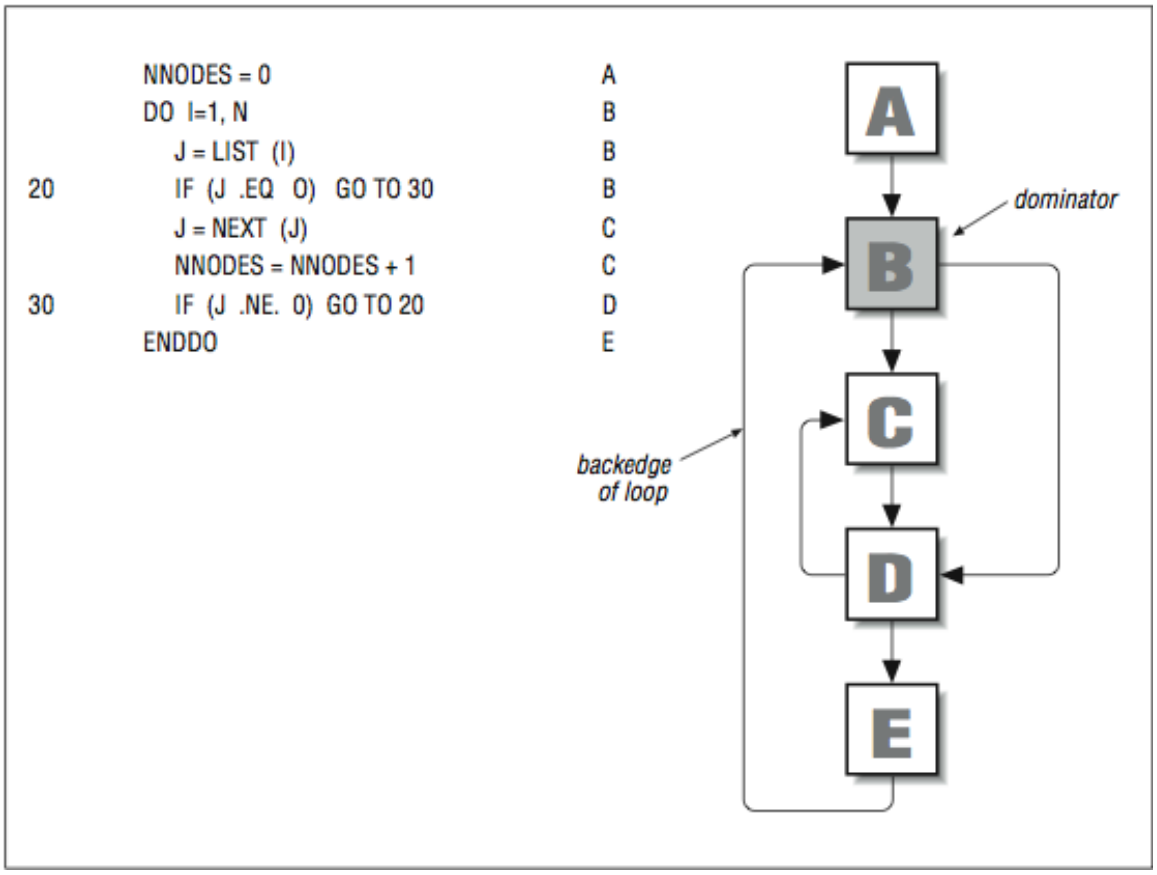


Figure 4.10

At first glance, it appears that the nodes C and D form a loop too. The problem is that C doesn't dominate D (and vice versa), because entry to either can be made from B, so condition 1 isn't satisfied. Generally, the flow graphs that come from code segments written with even the weakest appreciation for a structured design offer better loop candidates.

After identifying a loop, the compiler can concentrate on that portion of the flow graph, looking for instructions to remove or push to the outside. Certain types of subexpressions, such as those found in array index expressions, can be simplified if they change in a predictable fashion from one iteration to the next.

In the continuing quest for parallelism, loops are generally our best sources for large amounts of parallelism. However, loops also provide new opportunities for those parallelism-killing dependencies.

4.4 Loop-Carried Dependencies [6]

4.4.1 Loop-Carried Dependencies

The notion of data dependence is particularly important when we look at loops, the hub of activity inside numerical applications. A well-designed loop can produce millions of operations that can all be performed in parallel. However, a single misplaced dependency in the loop can force it all to be run in serial. So the stakes are higher when looking for dependencies in loops.

Some constructs are completely independent, right out of the box. The question we want to ask is "Can two different iterations execute at the same time, or is there a data dependency between them?" Consider the following loop:

```
DO I=1,N
  A(I) = A(I) + B(I)
ENDDO
```

For any two values of I and K, can we calculate the value of A(I) and A(K) at the same time? Below, we have manually unrolled several iterations of the previous loop, so they can be executed together:

```
A(I) = A(I) + B(I)
A(I+1) = A(I+1) + B(I+1)
A(I+2) = A(I+2) + B(I+2)
```

You can see that none of the results are used as an operand for another calculation. For instance, the calculation for A(I+1) can occur at the same time as the calculation for A(I) because the calculations are independent; you don't need the results of the first to determine the second. In fact, mixing up the order of the calculations won't change the results in the least. Relaxing the serial order imposed on these calculations makes it possible to execute this loop very quickly on parallel hardware.

4.4.1.1 Flow Dependencies

For comparison, look at the next code fragment:

[6] This content is available online at <http://cnx.org/content/m32782/1.1/>.

```
DO I=2,N
  A(I) = A(I-1) + B(I)
ENDDO
```

This loop has the regularity of the previous example, but one of the subscripts is changed. Again, it's useful to manually unroll the loop and look at several iterations together:

```
A(I) = A(I-1) + B(I)
A(I+1) = A(I) + B(I+1)
A(I+2) = A(I+1) + B(I+2)
```

In this case, there is a dependency problem. The value of `A(I+1)` depends on the value of `A(I)`, the value of `A(I+2)` depends on `A(I+1)`, and so on; every iteration depends on the result of a previous one. Dependencies that extend back to a previous calculation and perhaps a previous iteration (like this one), are loop carried *flow dependencies* or *backward dependencies*. You often see such dependencies in applications that perform Gaussian elimination on certain types of matrices, or numerical solutions to systems of differential equations. However, it is impossible to run such a loop in parallel (as written); the processor must wait for intermediate results before it can proceed.

In some cases, flow dependencies are impossible to fix; calculations are so dependent upon one another that we have no choice but to wait for previous ones to complete. Other times, dependencies are a function of the way the calculations are expressed. For instance, the loop above can be changed to reduce the dependency. By replicating some of the arithmetic, we can make it so that the second and third iterations depend on the first, but not on one another. The operation count goes up — we have an extra addition that we didn't have before — but we have reduced the dependency between iterations:

```
DO I=2,N,2
  A(I)   = A(I-1) + B(I)
  A(I+1) = A(I-1) + B(I) + B(I+1)
ENDDO
```

The speed increase on a workstation won't be great (most machines run the recast loop more slowly). However, some parallel computers can trade off additional calculations for reduced dependency and chalk up a net win.

4.4.1.2 Antidependencies

It's a different story when there is a loop-carried antidependency, as in the code below:

```
DO I=1,N
```

```
  A(I)   = B(I)   * E
  B(I)   = A(I+2) * C
ENDDO
```

In this loop, there is an antidependency between the variable A(I) and the variable A(I+2). That is, you must be sure that the instruction that uses A(I+2) does so before the previous one redefines it. Clearly, this is not a problem if the loop is executed serially, but remember, we are looking for opportunities to overlap instructions. Again, it helps to pull the loop apart and look at several iterations together. We have recast the loop by making many copies of the first statement, followed by copies of the second:

```
A(I)   = B(I)   * E
A(I+1) = B(I+1) * E
A(I+2) = B(I+2) * E
...
B(I)   = A(I+2) * C  ← assignment makes use of the new
B(I+1) = A(I+3) * C      value of A(I+2) incorrect.
B(I+2) = A(I+4) * C
```

The reference to `A(I+2)` needs to access an "old" value, rather than one of the new ones being calculated. If you perform all of the first statement followed by all of the second statement, the answers will be wrong. If you perform all of the second statement followed by all of the first statement, the answers will also be wrong. In a sense, to run the iterations in parallel, you must either save the `A` values to use for the second statement or store all of the `B` value in a temporary area until the loop completes.

We can also directly unroll the loop and find *some* parallelism:

```
1  A(I)   = B(I)   * E
2  B(I)   = A(I+2) * C →
3  A(I+1) = B(I+1) * E  | Output dependency
4  B(I+1) = A(I+3) * C  |
5  A(I+2) = B(I+2) * E ←
6  B(I+2) = A(I+4) * C
```

Statements 1–4 could all be executed simultaneously. Once those statements completed execution, statements 5–8 could execute in parallel. Using this approach, there are sufficient intervening statements between the dependent statements that we can see some parallel performance improvement from a superscalar RISC processor.

4.4.1.3 Output Dependencies

The third class of data dependencies, *output dependencies*, is of particular interest to users of parallel computers, particularly multiprocessors. Output dependencies involve getting the right values to the right variables when all calculations have been completed. Otherwise, an output dependency is violated. The loop below assigns new values to two elements of the vector A with each iteration:

```
DO I=1,N
  A(I)   = C(I) * 2.
  A(I+2) = D(I) + E
ENDDO
```

As always, we won't have any problems if we execute the code sequentially. But if several iterations are performed together, and statements are reordered, then incorrect values can be assigned to the last elements of `A`. For example, in the naive vectorized equivalent below, `A(I+2)` takes the wrong value because the assignments occur out of order:

```
A(I)   = C(I)   * 2.
A(I+1) = C(I+1) * 2.
A(I+2) = C(I+2) * 2.
A(I+2) = D(I)   + E ← Output dependency violated
A(I+3) = D(I+1) + E
A(I+4) = D(I+2) + E
```

Whether or not you have to worry about output dependencies depends on whether you are actually parallelizing the code. Your compiler will be conscious of the danger, and will be able to generate legal code — and possibly even fast code, if it's clever enough. But output dependencies occasionally become a problem for programmers.

4.4.1.4 Dependencies Within an Iteration

We have looked at dependencies that cross iteration boundaries but we haven't looked at dependencies within the same iteration. Consider the following code fragment:

```
DO I = 1,N
  D = B(I) * 17
  A(I) = D + 14
ENDDO
```

When we look at the loop, the variable `D` has a flow dependency. The second statement cannot start until the first statement has completed. At first glance this might appear to limit parallelism significantly. When we look closer and manually unroll several iterations of the loop, the situation gets worse:

```
D = B(I) * 17
A(I) = D + 14
D = B(I+1) * 17
```

```
A(I+1) = D + 14
D = B(I+2) * 17
A(I+2) = D + 14
```

Now, the variable `D` has flow, output, and antidependencies. It looks like this loop has no hope of running in parallel. However, there is a simple solution to this problem at the cost of some extra memory space, using a technique called *promoting a scalar to a vector*. We define `D` as an array `withN` elements and rewrite the code as follows:

```
DO I = 1,N
  D(I) = B(I) * 17
  A(I) = D(I) + 14
ENDDO
```

Now the iterations are all independent and can be run in parallel. Within each iteration, the first statement must run before the second statement.

4.4.1.5 Reductions

The sum of an array of numbers is one example of a *reduction* — so called because it reduces a vector to a scalar. The following loop to determine the total of the values in an array certainly looks as though it might be able to be run in parallel:

```
SUM = 0.0
DO I=1,N
  SUM = SUM + A(I)
ENDDO
```

However, if we perform our unrolling trick, it doesn't look very parallel:

```
SUM = SUM + A(I)
SUM = SUM + A(I+1)
SUM = SUM + A(I+2)
```

This loop also has all three types of dependencies and looks impossible to parallelize. If we are willing to accept the potential effect of rounding, we can add some parallelism to this loop as follows (again we did not add the preconditioning loop):

```
SUM0 = 0.0
```

```
SUM1 = 0.0
SUM2 = 0.0
SUM3 = 0.0
DO I=1,N,4
  SUM0 = SUM0 + A(I)
  SUM1 = SUM1 + A(I+1)
  SUM2 = SUM2 + A(I+2)
  SUM3 = SUM3 + A(I+3)
ENDDO
SUM = SUM0 + SUM1 + SUM2 + SUM3
```

Again, this is not precisely the same computation, but all four partial sums can be computed independently. The partial sums are combined at the end of the loop.

Loops that look for the maximum or minimum elements in an array, or multiply all the elements of an array, are also reductions. Likewise, some of these can be reorganized into partial results, as with the sum, to expose more computations. Note that the maximum and minimum are associative operators, so the results of the reorganized loop are identical to the sequential loop.

4.5 Ambiguous References[7]

4.5.1 Ambiguous References

Every dependency we have looked at so far has been clear cut; you could see exactly what you were dealing with by looking at the source code. But other times, describing a dependency isn't so easy. Recall this loop from the "Antidependencies" section earlier in this chapter:

```
DO I=1,N
  A(I) = B(I) * E
  B(I) = A(I+2) * C
ENDDO
```

Because each variable reference is solely a function of the index, `I`, it's clear what kind of dependency we are dealing with. Furthermore, we can describe how far apart (in iterations) a variable reference is from its definition. This is called the *dependency distance*. A negative value represents a flow dependency; a positive value means there is an antidependency. A value of zero says that no dependency exists between the reference and the definition. In this loop, the dependency distance for `A` is `+2` iterations.

However, array subscripts may be functions of other variables besides the loop index. It may be difficult to tell the distance between the use and definition of a particular element. It may even be impossible to tell whether the dependency is a flow dependency or an antidependency, or whether a dependency exists at all. Consequently, it may be impossible to determine if it's safe to overlap execution of different statements, as in the following loop:

```
DO I=1,N
```

[7] This content is available online at <http://cnx.org/content/m32788/1.1/>.

```
  A(I) = B(I) * E
  B(I) = A(I+K) * C ← K unknown
ENDDO
```

If the loop made use of `A(I+K)`, where the value of `K` was unknown, we wouldn't be able to tell (at least by looking at the code) anything about the kind of dependency we might be facing. If `K` is zero, we have a dependency within the iteration and no loop-carried dependencies. If `K` is positive we have an antidependency with distance `K`. Depending on the value for `K`, we might have enough parallelism for a superscalar processor. If `K` is negative, we have a loop-carried flow dependency, and we may have to execute the loop serially.

Ambiguous references, like `A(I+K)` above, have an effect on the parallelism we can detect in a loop. From the compiler perspective, it may be that this loop does contain two independent calculations that the author whimsically decided to throw into a single loop. But when they appear together, the compiler has to treat them conservatively, as if they were interrelated. This has a big effect on performance. If the compiler has to assume that consecutive memory references may ultimately access the same location, the instructions involved cannot be overlapped. One other option is for the compiler to generate two versions of the loop and check the value for `K` at runtime to determine which version of the loop to execute.

A similar situation occurs when we use integer index arrays in a loop. The loop below contains only a single statement, but you can't be sure that any iteration is independent without knowing the contents of the `K` and `J` arrays:

```
DO I=1,N
  A(K(I)) = A(K(I)) + B(J(I)) * C
ENDDO
```

For instance, what if all of the values for `K(I)` were the same? This causes the same element of the array `A` to be rereferenced with each iteration! That may seem ridiculous to you, but the compiler can't tell.

With code like this, it's common for every value of `K(I)` to be unique. This is called a *permutation*. If you can tell a compiler that it is dealing with a permutation, the penalty is lessened in some cases. Even so, there is insult being added to injury. Indirect references require more memory activity than direct references, and this slows you down.

4.5.1.1 Pointer Ambiguity in Numerical C Applications

FORTRAN compilers depend on programmers to observe aliasing rules. That is, programmers are not supposed to modify locations through pointers that may be aliases of one another. They can become aliases in several ways, such as when two dummy arguments receive pointers to the same storage locations:

```
CALL BOB (A,A)
  ...
END
SUBROUTINE BOB (X,Y) ← X,Y become aliases
```

C compilers don't enjoy the same restrictions on aliasing. In fact, there are cases where aliasing could be desirable. Additionally, C is blessed with pointer types, increasing the opportunities for aliasing to occur.

This means that a C compiler has to approach operations through pointers more conservatively than a FORTRAN compiler would. Let's look at some examples to see why.

The following loop nest looks like a FORTRAN loop cast in C. The arrays are declared or allocated all at once at the top of the routine, and the starting address and leading dimensions are visible to the compiler. This is important because it means that the storage relationship between the array elements is well known. Hence, you could expect good performance:

```
#define N ...
double *a[N][N], c[N][N], d;
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i][j] = a[i][j] + c[j][i] * d;
```

Now imagine what happens if you allocate the rows dynamically. This makes the address calculations more complicated. The loop nest hasn't changed; however, there is no guaranteed stride that can get you from one row to the next. This is because the storage relationship between the rows is unknown:

```
#define N ...
double *a[N], *c[N], d;
    for (i=0; i<N; i++) {
        a[i] = (double *) malloc (N*sizeof(double));
        c[i] = (double *) malloc (N*sizeof(double));
    }
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            a[i][j] = a[i][j] + c[j][i] * d;
```

In fact, your compiler knows even less than you might expect about the storage relationship. For instance, how can it be sure that references to a and c aren't aliases? It may be obvious to you that they're not. You might point out that *malloc* never overlaps storage. But the compiler isn't free to assume that. Who knows? You may be substituting your own version of *malloc*!

Let's look at a different example, where storage is allocated all at once, though the declarations are not visible to all routines that are using it. The following subroutine bob performs the same computation as our previous example. However, because the compiler can't see the declarations for a and c (they're in the main routine), it doesn't have enough information to be able to overlap memory references from successive iterations; the references could be aliases:

```
#define N...
main()
{
    double a[N][N], c[N][N], d;
    ...
    bob (a,c,d,N);
```

```
}
bob (double *a,double *c,double d,int n)
{
    int i,j;
    double *ap, *cp;
    for (i=0;i<n;i++) {
        ap = a + (i*n);
        cp = c + i;
        for (j=0; j<n; j++)
            *(ap+j) = *(ap+j) + *(cp+(j*n)) * d;
    }
}
```

To get the best performance, make available to the compiler as many details about the size and shape of your data structures as possible. Pointers, whether in the form of formal arguments to a subroutine or explicitly declared, can hide important facts about how you are using memory. The more information the compiler has, the more it can overlap memory references. This information can come from compiler directives or from making declarations visible in the routines where performance is most critical.

4.6 Closing Notes[8]

4.6.1 Closing Notes

You already knew there was a limit to the amount of parallelism in any given program. Now you know why. Clearly, if a program had no dependencies, you could execute the whole thing at once, given suitable hardware. But programs aren't infinitely parallel; they are often hardly parallel at all. This is because they contain dependencies of the types we saw above.

When we are writing and/or tuning our loops, we have a number of (sometimes conflicting) goals to keep in mind:

- Balance memory operations and computations.
- Minimize unnecessary operations.
- Access memory using unit stride if at all possible.
- Allow all of the loop iterations to be computed in parallel.

In the coming chapters, we will begin to learn more about executing our programs on parallel multiprocessors. At some point we will escape the bonds of compiler automatic optimization and begin to explicitly code the parallel portions of our code.

To learn more about compilers and dataflow, read The Art of Compiler Design: Theory and Practice by Thomas Pittman and James Peters (Prentice-Hall).

4.7 Exercises[9]

4.7.1 Exercises

Exercise 4.1

Identify the dependencies (if there are any) in the following loops. Can you think of ways to organize each loop for more parallelism?

[8] This content is available online at <http://cnx.org/content/m32789/1.1/>.
[9] This content is available online at <http://cnx.org/content/m32792/1.1/>.

a.

```
DO I=1,N-2
  A(I+2) = A(I) + 1.
ENDDO
```

b.

```
DO I=1,N-1,2
  A(I+1) = A(I) + 1.
ENDDO
```

c.

```
DO I=2,N
  A(I) = A(I-1) * 2.
  B = A(I-1)
ENDDO
```

d.

```
DO I=1,N
  IF(N .GT. M)
    A(I) = 1.
ENDDO
```

e.

```
DO I=1,N
  A(I,J) = A(I,K) + B
ENDDO
```

f.

```
DO I=1,N-1
  A(I+1,J) = A(I,K) + B
ENDDO
```

g.

```
for (i=0; i<n; i++)
  a[i] = b[i];
```

Exercise 4.2
Imagine that you are a parallelizing compiler, trying to generate code for the loop below. Why are references to A a challenge? Why would it help to know that K is equal to zero? Explain how you could partially vectorize the statements involving A if you knew that K had an absolute value of at least 8.

```
DO I=1,N
  E(I,M) = E(I-1,M+1) - 1.0
  B(I) = A(I+K) * C
  A(I) = D(I) * 2.0
ENDDO
```

Exercise 4.3
The following three statements contain a flow dependency, an antidependency and an output dependency. Can you identify each? Given that you are allowed to reorder the statements, can you find a permutation that produces the same values for the variables `C` and `B`? Show how you can reduce the dependencies by combining or rearranging calculations and using temporary variables.

```
B = A + C
B = C + D
C = B + D
```

Chapter 5

Shared-Memory Multiprocessors

5.1 Introduction[1]

5.1.1 Shared-Memory Multiprocessors

In the mid-1980s, shared-memory multiprocessors were pretty expensive and pretty rare. Now, as hardware costs are dropping, they are becoming commonplace. Many home computer systems in the under-$3000 range have a socket for a second CPU. Home computer operating systems are providing the capability to use more than one processor to improve system performance. Rather than specialized resources locked away in a central computing facility, these shared-memory processors are often viewed as a logical extension of the desktop. These systems run the same operating system (UNIX or NT) as the desktop and many of the same applications from a workstation will execute on these multiprocessor servers.

Typically a workstation will have from 1 to 4 processors and a server system will have 4 to 64 processors. Shared-memory multiprocessors have a significant advantage over other multiprocessors because all the processors share the same view of the memory, as shown in Figure 10-1.

These processors are also described as *uniform memory access* (also known as UMA) systems. This designation indicates that memory is equally accessible to all processors with the same performance.

The popularity of these systems is not due simply to the demand for high performance computing. These systems are excellent at providing high throughput for a multiprocessing load, and function effectively as high-performance database servers, network servers, and Internet servers. Within limits, their throughput is increased linearly as more processors are added.

In this book we are not so interested in the performance of database or Internet servers. That is too passé; buy more processors, get better throughput. We are interested in pure, raw, unadulterated compute speed for *our* high performance application. Instead of running hundreds of small jobs, we want to utilize all $750,000 worth of hardware for our single job.

The challenge is to find techniques that make a program that takes an hour to complete using one processor, complete in less than a minute using 64 processors. This is not trivial. Throughout this book so far, we have been on an endless quest for parallelism. In this and the remaining chapters, we will begin to see the payoff for all of your hard work and dedication!

The cost of a shared-memory multiprocessor can range from $4000 to $30 million. Some example systems include multiple-processor Intel systems from a wide range of vendors, SGI Power Challenge Series, HP/Convex C-Series, DEC AlphaServers, Cray vector/parallel processors, and Sun Enterprise systems. The SGI Origin 2000, HP/Convex Exemplar, Data General AV-20000, and Sequent NUMAQ-2000 all are uniform-memory, symmetric multiprocessing systems that can be linked to form even larger shared nonuniform memory-access systems. Among these systems, as the price increases, the number of CPUs increases, the performance of individual CPUs increases, and the memory performance increases.

[1] This content is available online at <http://cnx.org/content/m32797/1.1/>.

In this chapter we will study the hardware and software environment in these systems and learn how to execute our programs on these systems.

5.2 Symmetric Multiprocessing Hardware[2]

5.2.1 Symmetric Multiprocessing Hardware

In Figure 5.1 (Figure 10-1: A shared-memory multiprocessor), we viewed an ideal shared-memory multiprocessor. In this section, we look in more detail at how such a system is actually constructed. The primary advantage of these systems is the ability for any CPU to access all of the memory and peripherals. Furthermore, the systems need a facility for deciding among themselves who has access to what, and when, which means there will have to be hardware support for arbitration. The two most common architectural underpinnings for symmetric multiprocessing are *buses* and *crossbars*. The bus is the simplest of the two approaches. Figure 5.2 shows processors connected using a bus. A bus can be thought of as a set of parallel wires connecting the components of the computer (CPU, memory, and peripheral controllers), a set of protocols for communication, and some hardware to help carry it out. A bus is less expensive to build, but because all traffic must cross the bus, as the load increases, the bus eventually becomes a performance bottleneck.

Figure 10-1: A shared-memory multiprocessor

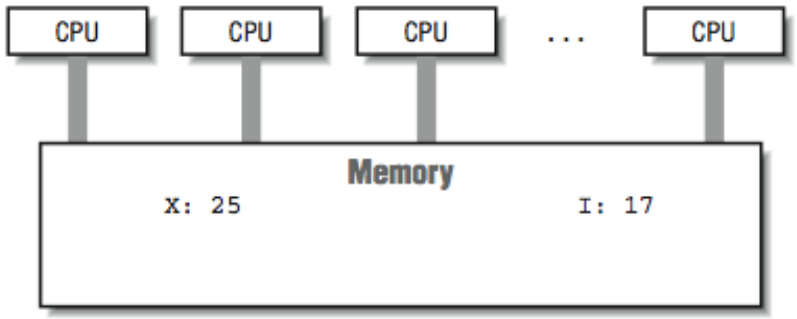


Figure 5.1

---

[2]This content is available online at <http://cnx.org/content/m32794/1.1/>.

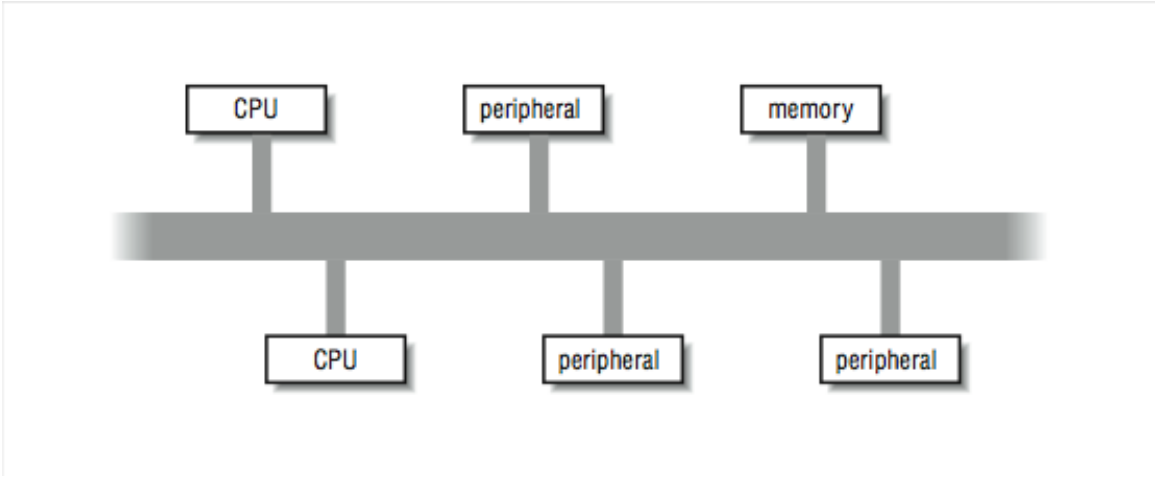


Figure 5.2: Figure 10-2: A typical bus architecture

A crossbar is a hardware approach to eliminate the bottleneck caused by a single bus. A crossbar is like several buses running side by side with attachments to each of the modules on the machine — CPU, memory, and peripherals. Any module can get to any other by a path through the crossbar, and multiple paths may be active simultaneously. In the 4×5 crossbar of Figure 5.3, for instance, there can be four active data transfers in progress at one time. In the diagram it looks like a patchwork of wires, but there is actually quite a bit of hardware that goes into constructing a crossbar. Not only does the crossbar connect parties that wish to communicate, but it must also actively arbitrate between two or more CPUs that want access to the same memory or peripheral. In the event that one module is too popular, it's the crossbar that decides who gets access and who doesn't. Crossbars have the best performance because there is no single shared bus. However, they are more expensive to build, and their cost increases as the number of ports is increased. Because of their cost, crossbars typically are only found at the high end of the price and performance spectrum.

Whether the system uses a bus or crossbar, there is only so much memory bandwidth to go around; four or eight processors drawing from one memory system can quickly saturate all available bandwidth. All of the techniques that improve memory performance (as described in Chapter 3, *Memory*) also apply here in the design of the memory subsystems attached to these buses or crossbars.

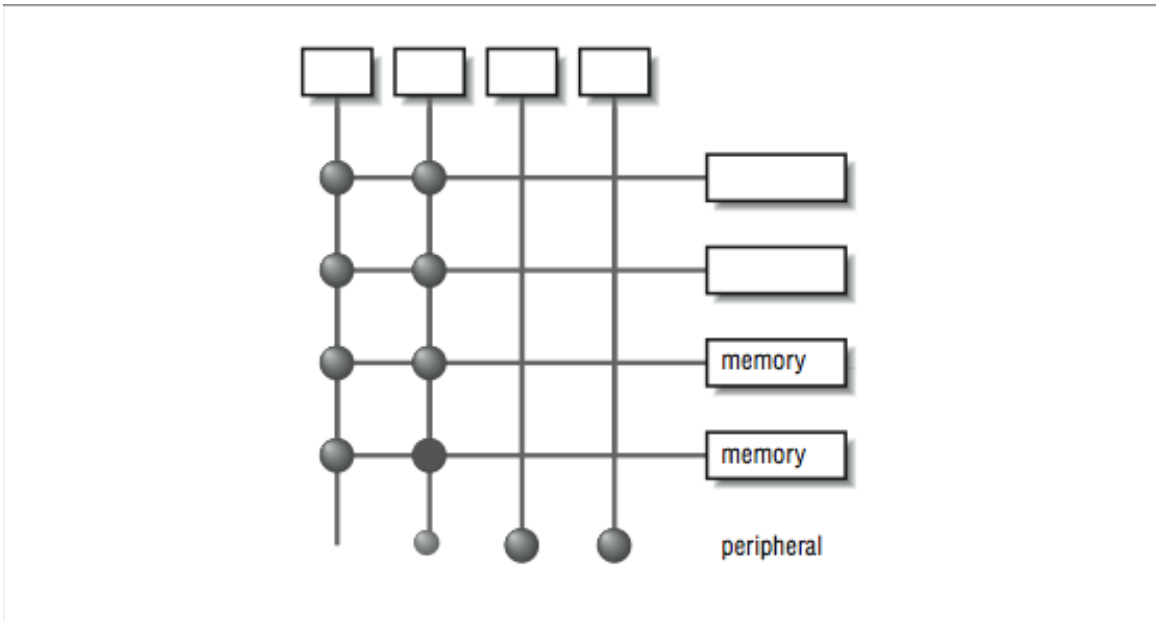


Figure 5.3: Figure 10-3: A crossbar

5.2.1.1 The Effect of Cache

The most common multiprocessing system is made up of commodity processors connected to memory and peripherals through a bus. Interestingly, the fact that these processors make use of cache somewhat mitigates the bandwidth bottleneck on a bus-based architecture. By connecting the processor to the cache and viewing the main memory through the cache, we significantly reduce the memory traffic across the bus. In this architecture, most of the memory accesses across the bus take the form of cache line loads and flushes. To understand why, consider what happens when the cache hit rate is very high. In Figure 5.4, a high cache hit rate eliminates some of the traffic that would have otherwise gone out across the bus or crossbar to main memory. Again, it is the notion of "locality of reference" that makes the system work. If you assume that a fair number of the memory references will hit in the cache, the equivalent attainable main memory bandwidth is more than the bus is actually capable of. This assumption explains why multiprocessors are designed with less bus bandwidth than the sum of what the CPUs can consume at once.

Imagine a scenario where two CPUs are accessing different areas of memory using unit stride. Both CPUs access the first element in a cache line at the same time. The bus arbitrarily allows one CPU access to the memory. The first CPU fills a cache line and begins to process the data. The instant the first CPU has completed its cache line fill, the cache line fill for the second CPU begins. Once the second cache line fill has completed, the second CPU begins to process the data in its cache line. If the time to process the data in a cache line is longer than the time to fill a cache line, the cache line fill for processor two completes before the next cache line request arrives from processor one. Once the initial conflict is resolved, both processors appear to have conflict-free access to memory for the remainder of their unit-stride loops.

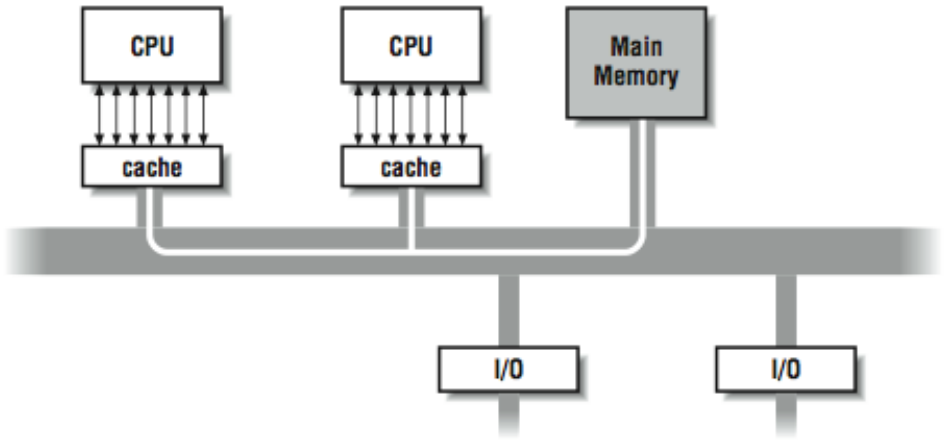


Figure 5.4: Figure 10-4: High cache hit rate reduces main memory traffic

In actuality, on some of the fastest bus-based systems, the memory bus is sufficiently fast that up to 20 processors can access memory using unit stride with very little conflict. If the processors are accessing memory using non-unit stride, bus and memory bank conflict becomes apparent, with fewer processors.

This bus architecture combined with local caches is very popular for general-purpose multiprocessing loads. The memory reference patterns for database or Internet servers generally consist of a combination of time periods with a small working set, and time periods that access large data structures using unit stride. Scientific codes tend to perform more non-unit-stride access than general-purpose codes. For this reason, the most expensive parallel-processing systems targeted at scientific codes tend to use crossbars connected to multibanked memory systems.

The main memory system is better shielded when a larger cache is used. For this reason, multiprocessors sometimes incorporate a two-tier cache system, where each processor uses its own small on-chip local cache, backed up by a larger second board-level cache with as much as 4 MB of memory. Only when neither can satisfy a memory request, or when data has to be written back to main memory, does a request go out over the bus or crossbar.

5.2.1.2 Coherency

Now, what happens when one CPU of a multiprocessor running a single program in parallel changes the value of a variable, and another CPU tries to read it? Where does the value come from? These questions are interesting because there can be multiple copies of each variable, and some of them can hold old or stale values.

For illustration, say that you are running a program with a shared variable A. Processor 1 changes the value of A and Processor 2 goes to read it.

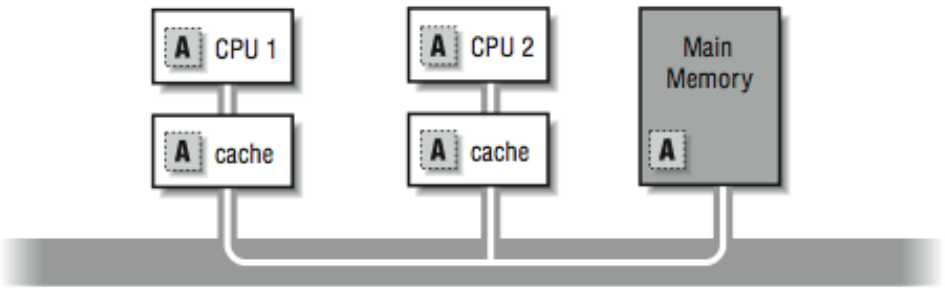


Figure 5.5: Figure 10-5: Multiple copies of variable A

In Figure 5.5, if Processor 1 is keeping `A` as a register-resident variable, then Processor 2 doesn't stand a chance of getting the correct value when it goes to look for it. There is no way that 2 can know the contents of 1's registers; so assume, at the very least, that Processor 1 writes the new value back out. Now the question is, where does the new value get stored? Does it remain in Processor 1's cache? Is it written to main memory? Does it get updated in Processor 2's cache?

Really, we are asking what kind of *cache coherency protocol* the vendor uses to assure that all processors see a uniform view of the values in "memory." It generally isn't something that the programmer has to worry about, except that in some cases, it can affect performance. The approaches used in these systems are similar to those used in single-processor systems with some extensions. The most straight-forward cache coherency approach is called a *write-through policy* : variables written into cache are simultaneously written into main memory. As the update takes place, other caches in the system see the main memory reference being performed. This can be done because all of the caches continuously monitor (also known as *snooping*) the traffic on the bus, checking to see if each address is in their cache. If a cache "notices" that it contains a copy of the data from the locations being written, it may either *invalidate* its copy of the variable or obtain new values (depending on the policy). One thing to note is that a write-through cache demands a fair amount of main memory bandwidth since each write goes out over the main memory bus. Furthermore, successive writes to the same location or bank are subject to the main memory cycle time and can slow the machine down.

A more sophisticated cache coherency protocol is called *copyback* or *writeback*. The idea is that you write values back out to main memory only when the cache housing them needs the space for something else. Updates of cached data are coordinated between the caches, by the caches, without help from the processor. Copyback caching also uses hardware that can monitor (snoop) and respond to the memory transactions of the other caches in the system. The benefit of this method over the write-through method is that memory traffic is reduced considerably. Let's walk through it to see how it works.

5.2.1.3 Cache Line States

For this approach to work, each cache must maintain a state for each line in its cache. The possible states used in the example include:

Modified: This cache line needs to be written back to memory.
Exclusive: There are no other caches that have this cache line.
Shared : There are read-only copies of this line in two or more caches.

Empty/Invalid: This cache line doesn't contain any useful data.

This particular coherency protocol is often called *MESI*. Other cache coherency protocols are more complicated, but these states give you an idea how multiprocessor writeback cache coherency works.

We start where a particular cache line is in memory and in none of the writeback caches on the systems. The first cache to ask for data from a particular part of memory completes a normal memory access; the main memory system returns data from the requested location in response to a cache miss. The associated cache line is marked *exclusive*, meaning that this is the only cache in the system containing a copy of the data; it is the owner of the data. If another cache goes to main memory looking for the same thing, the request is intercepted by the first cache, and the data is returned from the first cache — not main memory. Once an interception has occurred and the data is returned, the data is marked *shared* in both of the caches.

When a particular line is marked shared, the caches have to treat it differently than they would if they were the exclusive owners of the data — especially if any of them wants to modify it. In particular, a write to a shared cache entry is preceded by a broadcast message to all the other caches in the system. It tells them to invalidate their copies of the data. The one remaining cache line gets marked as *modified* to signal that it has been changed, and that it must be returned to main memory when the space is needed for something else. By these mechanisms, you can maintain cache coherence across the multiprocessor without adding tremendously to the memory traffic.

By the way, even if a variable is not shared, it's possible for copies of it to show up in several caches. On a symmetric multiprocessor, your program can bounce around from CPU to CPU. If you run for a little while on this CPU, and then a little while on that, your program will have operated out of separate caches. That means that there can be several copies of seemingly unshared variables scattered around the machine. Operating systems often try to minimize how often a process is moved between physical CPUs during context switches. This is one reason not to overload the available processors in a system.

5.2.1.4 Data Placement

There is one more pitfall regarding shared memory we have so far failed to mention. It involves data movement. Although it would be convenient to think of the multiprocessor memory as one big pool, we have seen that it is actually a carefully crafted system of caches, coherency protocols, and main memory. The problems come when your application causes lots of data to be traded between the caches. Each reference that falls out of a given processor's cache (especially those that require an update in another processor's cache) has to go out on the bus.

Often, it's slower to get memory from another processor's cache than from the main memory because of the protocol and processing overhead involved. Not only do we need to have programs with high locality of reference and unit stride, we also need to minimize the data that must be moved from one CPU to another.

5.3 Multiprocessor Software Concepts [3]

5.3.1 Multiprocessor Software Concepts

Now that we have examined the way shared-memory multiprocessor hardware operates, we need to examine how software operates on these types of computers. We still have to wait until the next chapters to begin making our FORTRAN programs run in parallel. For now, we use C programs to examine the fundamentals of multiprocessing and multithreading. There are several techniques used to implement multithreading, so the topics we will cover include:

- Operating system–supported multiprocessing
- User space multithreading
- Operating system-supported multithreading

The last of these is what we primarily will use to reduce the walltime of our applications.

[3] This content is available online at <http://cnx.org/content/m32800/1.1/>.

5.3.1.1 Operating System–Supported Multiprocessing

Most modern general-purpose operating systems support some form of multiprocessing. Multiprocessing doesn't require more than one physical CPU; it is simply the operating system's ability to run more than one *process* on the system. The operating system context-switches between each process at fixed time intervals, or on interrupts or input-output activity. For example, in UNIX, if you use the `ps` command, you can see the processes on the system:

```
% ps -a
  PID TTY      TIME CMD
28410 pts/34  0:00 tcsh
28213 pts/38  0:00 xterm
10488 pts/51  0:01 telnet
28411 pts/34  0:00 xbiff
11123 pts/25  0:00 pine
 3805 pts/21  0:00 elm
 6773 pts/44  5:48 ansys
   ...
% ps --a | grep ansys
  6773 pts/44 6:00 ansys
```

For each process we see the process identifier (PID), the terminal that is executing the command, the amount of CPU time the command has used, and the name of the command. The PID is unique across the entire system. Most UNIX commands are executed in a separate process. In the above example, most of the processes are waiting for some type of event, so they are taking very few resources except for memory. Process 6773[4] seems to be executing and using resources. Running *ps* again confirms that the CPU time is increasing for the *ansys* process:

```
% vmstat 5
procs     memory          page            disk         faults      cpu
r b w   swap  free re mf pi po fr de sr f0 s0 -- --  in   sy  cs us sy id
3 0 0 353624 45432  0  0  1  0  0  0  0  0  0  0  0 461 5626 354 91  9  0
3 0 0 353248 43960  0 22  0  0  0  0  0  0 14  0  0 518 6227 385 89 11  0
```

Running the *vmstat 5* command tells us many things about the activity on the system. First, there are three runnable processes. If we had one CPU, only one would actually be running at a given instant. To allow all three jobs to progress, the operating system time-shares between the processes. Assuming equal priority, each process executes about 1/3 of the time. However, this system is a two-processor system, so each process executes about 2/3 of the time. Looking across the *vmstat* output, we can see paging activity (*pi*, *po*), context switches (*cs*), overall user time (*us*), system time (*sy*), and idle time (*id*).

Each process can execute a completely different program. While most processes are completely independent, they can cooperate and share information using interprocess communication (pipes, sockets) or various operating system-supported shared-memory areas. We generally don't use multiprocessing on these shared-memory systems as a technique to increase single-application performance. We will explore techniques that use multiprocessing coupled with communication to improve performance on scalable parallel processing systems in Chapter 12, *Large- Scale Parallel Computing*.

[4] ANSYS is a commonly used structural-analysis package.

5.3.1.2 Multiprocessing software

In this section, we explore how programs access multiprocessing features.[5] In this example, the program creates a new process using the `fork( )` function. The new process (child) prints some messages and then changes its identity using `exec( )` by loading a new program. The original process (parent) prints some messages and then waits for the child process to complete:

```
int globvar;   /* A global variable */

main () {

 int pid,status,retval;
 int stackvar;   /* A stack variable */

 globvar = 1;
 stackvar = 1;
 printf("Main - calling fork globvar=%d stackvar=%d\n",globvar,stackvar);
 pid = fork();
 printf("Main - fork returned pid=%d\n",pid);
 if ( pid == 0 ) {
   printf("Child - globvar=%d stackvar=%d\n",globvar,stackvar);
   sleep(1);
   printf("Child - woke up globvar=%d stackvar=%d\n",globvar,stackvar);
   globvar = 100;
   stackvar = 100;
   printf("Child - modified globvar=%d stackvar=%d\n",globvar,stackvar);
   retval = execl("/bin/date", (char *) 0 );
   printf("Child - WHY ARE WE HERE retval=%d\n",retval);
 } else {
   printf("Parent - globvar=%d stackvar=%d\n",globvar,stackvar);
   globvar = 5;
   stackvar = 5;
   printf("Parent - sleeping globvar=%d stackvar=%d\n",globvar,stackvar);
   sleep(2);
   printf("Parent - woke up globvar=%d stackvar=%d\n",globvar,stackvar);
   printf("Parent - waiting for pid=%d\n",pid);
   retval = wait(&status);
   status = status ≫ 8; /* Return code in bits 15-8 */
   printf("Parent - status=%d retval=%d\n",status,retval);
 }
}
```

The key to understanding this code is to understand how the `fork( )` function operates. The simple summary is that the `fork( )` function is called once in a process and returns twice, once in the original process and once in a newly created process. The newly created process is an identical copy of the original process. All the variables (local and global) have been duplicated. Both processes have access to all of the open files of the original process. Figure 5.6 (Figure 10-6: How a fork operates) shows how the fork operation creates a new process.

[5] These examples are written in C using the POSIX 1003.1 application programming interface. This example runs on most UNIX systems and on other POSIX-compliant systems including OpenNT, Open- VMS, and many others.

The only difference between the processes is that the return value from the `fork( )` function call is 0 in the new (child) process and the process identifier (shown by the `ps` command) in the original (parent) process. This is the program output:

```
recs % cc -o fork fork.c
recs % fork
Main - calling fork globvar=1 stackvar=1
Main - fork returned pid=19336
Main - fork returned pid=0
Parent - globvar=1 stackvar=1
Parent - sleeping globvar=5 stackvar=5
Child - globvar=1 stackvar=1
Child - woke up globvar=1 stackvar=1
Child - modified globvar=100 stackvar=100
Thu Nov 6 22:40:33
Parent - woke up globvar=5 stackvar=5
Parent - waiting for pid=19336
Parent - status=0 retval=19336
recs %
```

Tracing this through, first the program sets the global and stack variable to one and then calls `fork( )`. During the `fork( )` call, the operating system suspends the process, makes an exact duplicate of the process, and then restarts both processes. You can see two messages from the statement immediately after the fork. The first line is coming from the original process, and the second line is coming from the new process. If you were to execute a `ps` command at this moment in time, you would see two processes running called "fork." One would have a process identifier of 19336.

Figure 10-6: How a fork operates

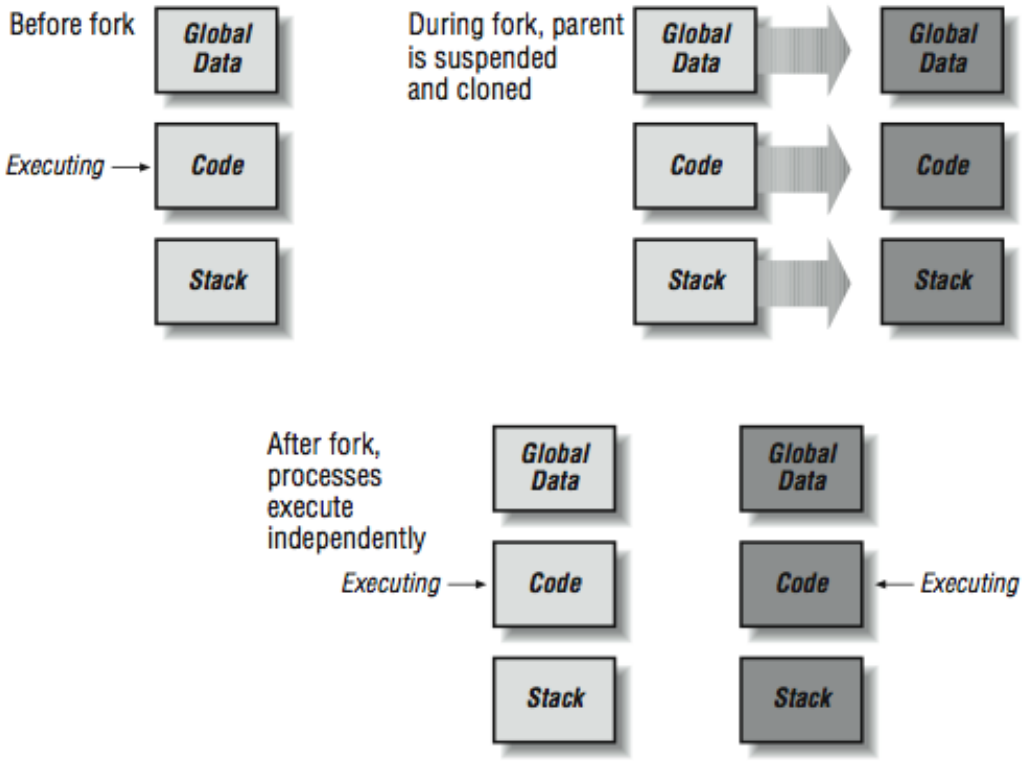


Figure 5.6

As both processes start, they execute an IF-THEN-ELSE and begin to perform different actions in the parent and child. Notice that *globvar* and *stackvar* are set to 5 in the parent, and then the parent sleeps for two seconds. At this point, the child begins executing. The values for *globvar* and *stackvar* are unchanged in the child process. This is because these two processes are operating in completely independent memory spaces. The child process sleeps for one second and sets its copies of the variables to 100. Next, the child process calls the `execl( )` function to overwrite its memory space with the UNIX date program. Note that the `execl( )` never returns; the date program takes over all of the resources of the child process. If you were to do a *ps* at this moment in time, you still see two processes on the system but process 19336 would be called "date." The date command executes, and you can see its output.[6]

The parent wakes up after a brief two-second sleep and notices that its copies of global and local variables have not been changed by the action of the child process. The parent then calls the `wait( )` function to

[6]It's not uncommon for a human parent process to "fork" and create a human child process that initially seems to have the same identity as the parent. It's also not uncommon for the child process to change its overall identity to be something very different from the parent at some later point. Usually human children wait 13 years or so before this change occurs, but in UNIX, this happens in a few microseconds. So, in some ways, in UNIX, there are many parent processes that are "disappointed" because their children did not turn out like them!

determine if any of its children exited. The `wait( )` function returns which child has exited and the status code returned by that child process (in this case, process 19336).

5.3.1.3 User Space Multithreading

A *thread* is different from a process. When you add threads, they are added to the existing process rather than starting in a new process. Processes start with a single thread of execution and can add or remove threads throughout the duration of the program. Unlike processes, which operate in different memory spaces, all threads in a process share the same memory space. Figure 10-7 shows how the creation of a thread differs from the creation of a process. Not all of the memory space in a process is shared between all threads. In addition to the global area that is shared across all threads, each thread has a *thread private* area for its own local variables. It's important for programmers to know when they are working with shared variables and when they are working with local variables.

When attempting to speed up high performance computing applications, threads have the advantage over processes in that multiple threads can cooperate and work on a shared data structure to hasten the computation. By dividing the work into smaller portions and assigning each smaller portion to a separate thread, the total work can be completed more quickly.

Multiple threads are also used in high performance database and Internet servers to improve the overall *throughput* of the server. With a single thread, the program can either be waiting for the next network request or reading the disk to satisfy the previous request. With multiple threads, one thread can be waiting for the next network transaction while several other threads are waiting for disk I/O to complete.

The following is an example of a simple multithreaded application.[7] It begins with a single master thread that creates three additional threads. Each thread prints some messages, accesses some global and local variables, and then terminates:

```
#define_REENTRANT                /* basic lines for threads */
#include <stdio.h>
#include <pthread.h>

#define THREAD_COUNT 3
void *TestFunc(void *);
int globvar;                      /* A global variable */
int index[THREAD_COUNT]           /* Local zero-based thread index */
pthread_t thread_id[THREAD_COUNT];  /* POSIX Thread IDs */

main() {
  int i,retval;
  pthread_t tid;

  globvar = 0;
  printf("Main - globvar=%d\n",globvar);
  for(i=0;i<THREAD_COUNT;i++) {
    index[i] = i;
    retval = pthread_create(&tid,NULL,TestFunc,(void *) index[i]);
    printf("Main - creating i=%d tid=%d retval=%d\n",i,tid,retval);
    thread_id[i] = tid;
  }
```

[7] This example uses the IEEE POSIX standard interface for a thread library. If your system supports POSIX threads, this example should work. If not, there should be similar routines on your system for each of the thread functions.

```
printf("Main thread - threads started globvar=%d\n",globvar);
for(i=0;i<THREAD_COUNT;i++) {
  printf("Main - waiting for join %d\n",thread_id[i]);
  retval = pthread_join( thread_id[i], NULL ) ;
  printf("Main - back from join %d retval=%d\n",i,retval);
  }
  printf("Main thread - threads completed globvar=%d\n",globvar);
}

void *TestFunc(void *parm) {
  int me,self;

  me = (int) parm; /* My own assigned thread ordinal */
  self = pthread_self(); /* The POSIX Thread library thread number */
  printf("TestFunc me=%d - self=%d globvar=%d\n",me,self,globvar);
  globvar = me + 15;
  printf("TestFunc me=%d - sleeping globvar=%d\n",me,globvar);
  sleep(2);
  printf("TestFunc me=%d - done param=%d globvar=%d\n",me,self,globvar);
}
```

Figure 10-7: Creating a thread

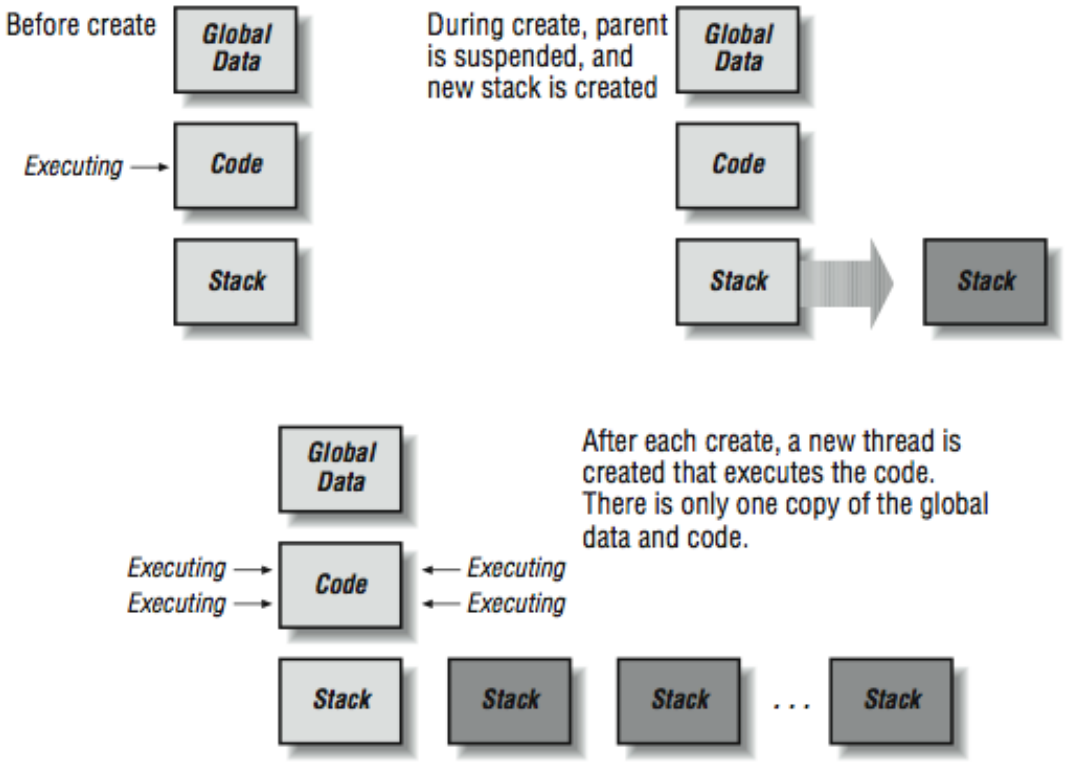


Figure 5.7

The global shared areas in this case are those variables declared in the static area outside the `main( )` code. The local variables are any variables declared within a routine. When threads are added, each thread gets its own function call stack. In C, the *automatic* variables that are declared at the beginning of each routine are allocated on the stack. As each thread enters a function, these variables are separately allocated on that particular thread's stack. So these are the *thread-local* variables.

Unlike the `fork( )` function, the `pthread_create( )` function creates a new thread, and then control is returned to the calling thread. One of the parameters of the `pthread_create( )` is the name of a function.

New threads begin execution in the function `TestFunc( )` and the thread finishes when it returns from this function. When this program is executed, it produces the following output:

```
recs % cc -o create1 -lpthread -lposix4 create1.c
recs % create1
```

```
Main - globvar=0
Main - creating i=0 tid=4 retval=0
Main - creating i=1 tid=5 retval=0
Main - creating i=2 tid=6 retval=0
Main thread - threads started globvar=0
Main - waiting for join 4
TestFunc me=0 - self=4 globvar=0
TestFunc me=0 - sleeping globvar=15
TestFunc me=1 - self=5 globvar=15
TestFunc me=1 - sleeping globvar=16
TestFunc me=2 - self=6 globvar=16
TestFunc me=2 - sleeping globvar=17
TestFunc me=2 - done param=6 globvar=17
TestFunc me=1 - done param=5 globvar=17
TestFunc me=0 - done param=4 globvar=17
Main - back from join 0 retval=0
Main - waiting for join 5
Main - back from join 1 retval=0
Main - waiting for join 6
Main - back from join 2 retval=0
Main thread -- threads completed globvar=17
recs %
```

You can see the threads getting created in the loop. The master thread completes the `pthread_create( )` loop, executes the second loop, and calls the `pthread_join( )` function. This function suspends the master thread until the specified thread completes. The master thread is waiting for Thread 4 to complete. Once the master thread suspends, one of the new threads is started. Thread 4 starts executing. Initially the variable `globvar` is set to 0 from the main program. The `self, me`, and `param` variables are thread-local variables, so each thread has its own copy. Thread 4 sets `globvar` to 15 and goes to sleep. Then Thread 5 begins to execute and sees `globvar` set to 15 from Thread 4; Thread 5 sets `globvar` to 16, and goes to sleep. This activates Thread 6, which sees the current value for `globvar` and sets it to 17. Then Threads 6, 5, and 4 wake up from their sleep, all notice the latest value of 17 in `globvar`, and return from the `TestFunc( )` routine, ending the threads.

All this time, the master thread is in the middle of `a pthread_join( )` waiting for Thread 4 to complete. As Thread 4 completes, the `pthread_join( )` returns. The master thread then calls `pthread_join( )` repeatedly to ensure that all three threads have been completed. Finally, the master thread prints out the value for `globvar` that contains the latest value of 17.

To summarize, when an application is executing with more than one thread, there are shared global areas and thread private areas. Different threads execute at different times, and they can easily work together in shared areas.

5.3.1.4 Limitations of user space multithreading

Multithreaded applications were around long before multiprocessors existed. It is quite practical to have multiple threads with a single CPU. As a matter of fact, the previous example would run on a system with any number of processors, including one. If you look closely at the code, it performs a sleep operation at each critical point in the code. One reason to add the sleep calls is to slow the program down enough that you can actually see what is going on. However, these sleep calls also have another effect. When one thread enters the sleep routine, it causes the thread library to search for other "runnable" threads. If a runnable thread is found, it begins executing immediately while the calling thread is "sleeping." This is called a *user-space thread context* switch. The process actually has one operating system thread shared among several logical

user threads. When library routines (such as *sleep*) are called, the thread library[8] jumps in and reschedules threads.

We can explore this effect by substituting the following `SpinFunc( )` function, replacing `TestFunc( )` function in the `pthread_create( )` call in the previous example:

```
void *SpinFunc(void *parm) {
  int me;
  me = (int) parm;
  printf("SpinFunc me=%d - sleeping %d seconds ...\n", me, me+1);
  sleep(me+1);
  printf("SpinFunc me=%d -- wake globvar=%d...\n", me, globvar);
  globvar ++;
  printf("SpinFunc me=%d - spinning globvar=%d...\n", me, globvar);
  while(globvar < THREAD_COUNT ) ;
  printf("SpinFunc me=%d -- done globvar=%d...\n", me, globvar);
  sleep(THREAD_COUNT+1);
}
```

If you look at the function, each thread entering this function prints a message and goes to sleep for 1, 2, and 3 seconds. Then the function increments `globvar` (initially set to 0 in main) and begins a while-loop, continuously checking the value of `globvar.` As time passes, the second and third threads should finish their `sleep( )`, increment the value for `globvar`, and begin the while-loop. When the last thread reaches the loop, the value for `globvar` is 3 and all the threads exit the loop. However, this isn't what happens:

```
recs % create2 &
[1] 23921
recs %
Main - globvar=0
Main - creating i=0 tid=4 retval=0
Main - creating i=1 tid=5 retval=0
Main - creating i=2 tid=6 retval=0
Main thread - threads started globvar=0
Main - waiting for join 4
SpinFunc me=0 - sleeping 1 seconds ...
SpinFunc me=1 - sleeping 2 seconds ...
SpinFunc me=2 - sleeping 3 seconds ...
SpinFunc me=0 - wake globvar=0...
SpinFunc me=0 - spinning globvar=1...

recs % ps
  PID TTY      TIME CMD
 23921 pts/35  0:09 create2
recs % ps
  PID TTY      TIME CMD
```

[8] The pthreads library supports both user-space threads and operating-system threads, as we shall soon see. Another popular early threads package was called *cthreads*.

```
 23921 pts/35   1:16 create2
recs % kill -9 23921
[1]    Killed                  create2
recs %
```

We run the program in the background[9] and everything seems to run fine. All the threads go to sleep for 1, 2, and 3 seconds. The first thread wakes up and starts the loop waiting for `globvar` to be incremented by the other threads. Unfortunately, with user space threads, there is no automatic time sharing. Because we are in a CPU loop that never makes a system call, the second and third threads never get scheduled so they can complete their `sleep( )` call. To fix this problem, we need to make the following change to the code:

```
while(globvar < THREAD_COUNT ) sleep(1) ;
```

With this sleep[10] call, Threads 2 and 3 get a chance to be "scheduled." They then finish their sleep calls, increment the `globvar` variable, and the program terminates properly.

You might ask the question, "Then what is the point of user space threads?" Well, when there is a high performance database server or Internet server, the multiple logical threads can overlap network I/O with database I/O and other background computations. This technique is not so useful when the threads all want to perform simultaneous CPU-intensive computations. To do this, you need threads that are created, managed, and scheduled by the operating system rather than a user library.

5.3.1.5 Operating System-Supported Multithreading

When the operating system supports multiple threads per process, you can begin to use these threads to do simultaneous computational activity. There is still no requirement that these applications be executed on a multiprocessor system. When an application that uses four operating system threads is executed on a single processor machine, the threads execute in a time-shared fashion. If there is no other load on the system, each thread gets 1/4 of the processor. While there are good reasons to have more threads than processors for noncompute applications, it's not a good idea to have more active threads than processors for compute-intensive applications because of thread-switching overhead. (For more detail on the effect of too many threads, see Appendix D, How FORTRAN Manages Threads at Runtime.

If you are using the POSIX threads library, it is a simple modification to request that your threads be created as operating-system rather rather than user threads, as the following code shows:

```
#define _REENTRANT      /* basic 3-lines for threads */
#include <stdio.h>
#include <pthread.h>

#define THREAD_COUNT 2
void *SpinFunc(void *);
int globvar;                        /* A global variable */
int index[THREAD_COUNT];            /* Local zero-based thread index */
pthread_t thread_id[THREAD_COUNT];      /* POSIX Thread IDs */
pthread_attr_t attr;                /* Thread attributes NULL=use default */
```

[9] Because we know it will hang and ignore interrupts.

[10] Some thread libraries support a call to a routine sched_yield( ) that checks for runnable threads. If it finds a runnable thread, it runs the thread. If no thread is runnable, it returns immediately to the calling thread. This routine allows a thread that has the CPU to ensure that other threads make progress during CPU-intensive periods of its code.

```
main() {
  int i,retval;
  pthread_t tid;

  globvar = 0;
  pthread_attr_init(&attr);      /* Initialize attr with defaults */
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  printf("Main - globvar=%d\n",globvar);
  for(i=0;i<THREAD_COUNT;i++) {
    index[i] = i;
    retval = pthread_create(&tid,&attr,SpinFunc,(void *) index[i]);
    printf("Main - creating i=%d tid=%d retval=%d\n",i,tid,retval);
    thread_id[i] = tid;
  }
  printf("Main thread - threads started globvar=%d\n",globvar);
  for(i=0;i<THREAD_COUNT;i++) {
    printf("Main - waiting for join %d\n",thread_id[i]);
    retval = pthread_join( thread_id[i], NULL ) ;
    printf("Main - back from join %d retval=%d\n",i,retval);
  }
  printf("Main thread - threads completed globvar=%d\n",globvar);
}
```

The code executed by the master thread is modified slightly. We create an "attribute" data structure and set the `PTHREAD_SCOPE_SYSTEM` attribute to indicate that we would like our new threads to be created and scheduled by the operating system. We use the attribute information on the call to `pthread_create( )`. None of the other code has been changed. The following is the execution output of this new program:

```
recs % create3
Main - globvar=0
Main - creating i=0 tid=4 retval=0
SpinFunc me=0 - sleeping 1 seconds ...
Main - creating i=1 tid=5 retval=0
Main thread - threads started globvar=0
Main - waiting for join 4
SpinFunc me=1 - sleeping 2 seconds ...
SpinFunc me=0 - wake globvar=0...
SpinFunc me=0 - spinning globvar=1...
SpinFunc me=1 - wake globvar=1...
SpinFunc me=1 - spinning globvar=2...
SpinFunc me=1 - done globvar=2...
SpinFunc me=0 - done globvar=2...
Main - back from join 0 retval=0
Main - waiting for join 5
Main - back from join 1 retval=0
Main thread - threads completed globvar=2
recs %
```

Now the program executes properly. When the first thread starts spinning, the operating system is context switching between all three threads. As the threads come out of their `sleep( )`, they increment their shared variable, and when the final thread increments the shared variable, the other two threads instantly notice the new value (because of the cache coherency protocol) and finish the loop. If there are fewer than three CPUs, a thread may have to wait for a time-sharing context switch to occur before it notices the updated global variable.

With operating-system threads and multiple processors, a program can realistically break up a large computation between several independent threads and compute the solution more quickly. Of course this presupposes that the computation could be done in parallel in the first place.

5.4 Techniques for Multithreaded Programs[11]

5.4.1 Techniques for Multithreaded Programs

Given that we have multithreaded capabilities and multiprocessors, we must still convince the threads to work together to accomplish some overall goal. Often we need some ways to coordinate and cooperate between the threads. There are several important techniques that are used while the program is running with multiple threads, including:

- Fork-join (or create-join) programming
- Synchronization using a critical section with a lock, semaphore, or mutex
- Barriers

Each of these techniques has an overhead associated with it. Because these overheads are necessary to go parallel, we must make sure that we have sufficient work to make the benefit of parallel operation worth the cost.

5.4.1.1 Fork-Join Programming

This approach is the simplest method of coordinating your threads. As in the earlier examples in this chapter, a master thread sets up some global data structures that describe the tasks each thread is to perform and then use the `pthread_create( )` function to activate the proper number of threads. Each thread checks the global data structure using its thread-id as an index to find its task. The thread then performs the task and completes. The master thread waits at a `pthread_join( )` point, and when a thread has completed, it updates the global data structure and creates a new thread. These steps are repeated for each major iteration (such as a time-step) for the duration of the program:

```
for(ts=0;ts<10000;ts++) { /* Time Step Loop */
   /* Setup tasks */
   for (ith=0;ith<NUM_THREADS;ith++) pthread_create(..,work_routine,..)
   for (ith=0;ith<NUM_THREADS;ith++) pthread_join(...)
}
work_routine() {
   /* Perform Task */
  return;
}
```

[11] This content is available online at <http://cnx.org/content/m32802/1.1/>.

The shortcoming of this approach is the overhead cost associated with creating and destroying an operating system thread for a potentially very short task.

The other approach is to have the threads created at the beginning of the program and to have them communicate amongst themselves throughout the duration of the application. To do this, they use such techniques as critical sections or barriers.

5.4.1.2 Synchronization

Synchronization is needed when there is a particular operation to a shared variable that can only be performed by one processor at a time. For example, in previous `SpinFunc( )` examples, consider the line:

```
globvar++;
```

In assembly language, this takes at least three instructions:

```
LOAD   R1,globvar
ADD    R1,1
STORE  R1,globvar
```

What if `globvar` contained 0, Thread 1 was running, and, at the precise moment it completed the `LOAD` into Register R1 and before it had completed the `ADD` or `STORE` instructions, the operating system interrupted the thread and switched to Thread 2? Thread 2 catches up and executes all three instructions using its registers: loading 0, adding 1 and storing the 1 back into `globvar`. Now Thread 2 goes to sleep and Thread 1 is restarted at the `ADD` instruction. Register R1 for Thread 1 contains the previously loaded value of 0; Thread 1 adds 1 and then stores 1 into `globvar`. What is wrong with this picture? We meant to use this code to count the number of threads that have passed this point. Two threads passed the point, but because of a bad case of bad timing, our variable indicates only that one thread passed. This is because the increment of a variable in memory is not `atomic`. That is, halfway through the increment, something else can happen.

Another way we can have a problem is on a multiprocessor when two processors execute these instructions simultaneously. They both do the `LOAD`, getting 0. Then they both add 1 and store 1 back to memory.[12] Which processor actually got the honor of storing *their* 1 back to memory is simply a race.

We must have some way of guaranteeing that only one thread can be in these three instructions at the same time. If one thread has started these instructions, all other threads must wait to enter until the first thread has exited. These areas are called *critical sections*. On single-CPU systems, there was a simple solution to critical sections: you could turn off interrupts for a few instructions and then turn them back on. This way you could guarantee that you would get all the way through before a timer or other interrupt occurred:

```
INTOFF                // Turn off Interrupts
LOAD    R1,globvar
ADD     R1,1
STORE   R1,globvar
INTON                 // Turn on Interrupts
```

[12] Boy, this is getting pretty picky. How often will either of these events really happen? Well, if it crashes your airline reservation system every 100,000 transactions or so, that would be way too often.

However, this technique does not work for longer critical sections or when there is more than one CPU. In these cases, you need a lock, a semaphore, or a mutex. Most thread libraries provide this type of routine. To use a mutex, we have to make some modifications to our example code:

```
  ...
pthread_mutex_t my_mutex; /* MUTEX data structure */
  ...

main() {
  ...
  pthread_attr_init(&attr); /* Initialize attr with defaults */
  pthread_mutex_init (&my_mutex, NULL);
   .... pthread_create( ... )
    ...
}
void *SpinFunc(void *parm)    {
 ...
  pthread_mutex_lock (&my_mutex);
  globvar ++;
  pthread_mutex_unlock (&my_mutex);
  while(globvar < THREAD_COUNT ) ;
  printf("SpinFunc me=%d -- done globvar=%d...\n", me, globvar);
  ...
}
```

The mutex data structure must be declared in the shared area of the program. Before the threads are created, `pthread_mutex_init` must be called to initialize the mutex. Before `globvar` is incremented, we must lock the mutex and after we finish updating `globvar` (three instructions later), we unlock the mutex. With the code as shown above, there will never be more than one processor executing the `globvar++` line of code, and the code will never hang because an increment was missed. Semaphores and locks are used in a similar way.

Interestingly, when using user space threads, an attempt to lock an already locked mutex, semaphore, or lock can cause a thread context switch. This allows the thread that "owns" the lock a better chance to make progress toward the point where they will unlock the critical section. Also, the act of unlocking a mutex can cause the thread waiting for the mutex to be dispatched by the thread library.

5.4.1.3 Barriers

Barriers are different than critical sections. Sometimes in a multithreaded application, you need to have all threads arrive at a point before allowing any threads to execute beyond that point. An example of this is a *time-based simulation*. Each task processes its portion of the simulation but must wait until all of the threads have completed the current time step before any thread can begin the next time step. Typically threads are created, and then each thread executes a loop with one or more barriers in the loop. The rough pseudocode for this type of approach is as follows:

```
main() {
  for (ith=0;ith<NUM_THREADS;ith++) pthread_create(..,work_routine,..)
  for (ith=0;ith<NUM_THREADS;ith++) pthread_join(...) /* Wait a long time */
  exit()
}
```

```
work_routine() {

  for(ts=0;ts<10000;ts++) { /* Time Step Loop */
        /* Compute total forces on particles */
    wait_barrier();
        /* Update particle positions based on the forces */
    wait_barrier();
  }
  return;
}
```

In a sense, our `SpinFunc( )` function implements a barrier. It sets a variable initially to 0. Then as threads arrive, the variable is incremented in a critical section. Immediately after the critical section, the thread spins until the precise moment that all the threads are in the spin loop, at which time all threads exit the spin loop and continue on.

For a critical section, only one processor can be executing in the critical section at the same time. For a barrier, all processors must arrive at the barrier before any of the processors can leave.

5.5 A Real Example [13]

5.5.1 A Real Example

In all of the above examples, we have focused on the mechanics of shared memory, thread creation, and thread termination. We have used the `sleep( )` routine to slow things down sufficiently to see interactions between processes. But we want to go very fast, not just learn threading for threading's sake.

The example code below uses the multithreading techniques described in this chapter to speed up a sum of a large array. The *hpcwall* routine is from Chapter 6, *Timing and Profiling*.

This code allocates a four-million-element double-precision array and fills it with random numbers between 0 and 1. Then using one, two, three, and four threads, it sums up the elements in the array:

```
#define _REENTRANT        /* basic 3-lines for threads */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX_THREAD 4
void *SumFunc(void *);
int ThreadCount;                    /* Threads on this try */
double GlobSum;                     /* A global variable */
int index[MAX_THREAD];              /* Local zero-based thread index */
pthread_t thread_id[MAX_THREAD];    /* POSIX Thread IDs */
pthread_attr_t attr;                /* Thread attributes NULL=use default */
pthread_mutex_t my_mutex;           /* MUTEX data structure */

#define MAX_SIZE 4000000
double array[MAX_SIZE];             /* What we are summing... */
void hpcwall(double *);
```

[13] This content is available online at <http://cnx.org/content/m32804/1.1/>.

```
main() {
  int i,retval;
  pthread_t tid;
  double single,multi,begtime,endtime;

  /* Initialize things */
  for (i=0; i<MAX_SIZE; i++) array[i] = drand48();
  pthread_attr_init(&attr);     /* Initialize attr with defaults */
  pthread_mutex_init (&my_mutex, NULL);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* Single threaded sum */
  GlobSum = 0;
  hpcwall(&begtime);
  for(i=0; i<MAX_SIZE;i++) GlobSum = GlobSum + array[i];
  hpcwall(&endtime);
  single = endtime - begtime;
  printf("Single sum=%lf time=%lf\n",GlobSum,single);

  /* Use different numbers of threads to accomplish the same thing */
  for(ThreadCount=2;ThreadCount<=MAX_THREAD; ThreadCount++) {
    printf("Threads=%d\n",ThreadCount);
    GlobSum = 0;
    hpcwall(&begtime);
    for(i=0;i<ThreadCount;i++) {
      index[i] = i;
      retval = pthread_create(&tid,&attr,SumFunc,(void *) index[i]);
      thread_id[i] = tid;
    }
    for(i=0;i<ThreadCount;i++) retval = pthread_join(thread_id[i],NULL);
    hpcwall(&endtime);
    multi = endtime - begtime;
    printf("Sum=%lf time=%lf\n",GlobSum,multi);
    printf("Efficiency = %lf\n",single/(multi*ThreadCount));
  } /* End of the ThreadCount loop */
}

void *SumFunc(void *parm){
  int i,me,chunk,start,end;
  double LocSum;

  /* Decide which iterations belong to me */
  me = (int) parm;
  chunk = MAX_SIZE / ThreadCount;
  start = me * chunk;
  end = start + chunk; /* C-Style - actual element + 1 */
  if ( me == (ThreadCount-1) ) end = MAX_SIZE;
  printf("SumFunc me=%d start=%d end=%d\n",me,start,end);

  /* Compute sum of our subset*/
  LocSum = 0;
```

```
  for(i=start;i<end;i++ ) LocSum = LocSum + array[i];

  /* Update the global sum and return to the waiting join */
  pthread_mutex_lock (&my_mutex);
  GlobSum = GlobSum + LocSum;
  pthread_mutex_unlock (&my_mutex);
}
```

First, the code performs the sum using a single thread using a for-loop. Then for each of the parallel sums, it creates the appropriate number of threads that call `SumFunc( )`. Each thread starts in `SumFunc( )` and initially chooses an area to operation in the shared array. The "strip" is chosen by dividing the overall array up evenly among the threads with the last thread getting a few extra if the division has a remainder.

Then, each thread independently performs the sum on its area. When a thread has finished its computation, it uses a mutex to update the global sum variable with its contribution to the global sum:

```
recs % addup
Single sum=7999998000000.000000 time=0.256624
Threads=2
SumFunc me=0 start=0 end=2000000
SumFunc me=1 start=2000000 end=4000000
Sum=7999998000000.000000 time=0.133530
Efficiency = 0.960923
Threads=3
SumFunc me=0 start=0 end=1333333
SumFunc me=1 start=1333333 end=2666666
SumFunc me=2 start=2666666 end=4000000
Sum=7999998000000.000000 time=0.091018
Efficiency = 0.939829
Threads=4
SumFunc me=0 start=0 end=1000000
SumFunc me=1 start=1000000 end=2000000
SumFunc me=2 start=2000000 end=3000000
SumFunc me=3 start=3000000 end=4000000
Sum=7999998000000.000000 time=0.107473
Efficiency = 0.596950
recs %
```

There are some interesting patterns. Before you interpret the patterns, you must know that this system is a three-processor Sun Enterprise 3000. Note that as we go from one to two threads, the time is reduced to one-half. That is a good result given how much it costs for that extra CPU. We characterize how well the additional resources have been used by computing an efficiency factor that should be 1.0. This is computed by multiplying the wall time by the number of threads. Then the time it takes on a single processor is divided by this number. If you are using the extra processors well, this evaluates to 1.0. If the extra processors are used pretty well, this would be about 0.9. If you had two threads, and the computation did not speed up at all, you would get 0.5.

At two and three threads, wall time is dropping, and the efficiency is well over 0.9. However, at four threads, the wall time increases, and our efficiency drops very dramatically. This is because we now have more threads than processors. Even though we have four threads that could execute, they must be time-

sliced between three processors.[14] This is even worse that it might seem. As threads are switched, they move from processor to processor and their caches must also move from processor to processor, further slowing performance. This cache-thrashing effect is not too apparent in this example because the data structure is so large, most memory references are not to values previously in cache.

It's important to note that because of the nature of floating-point (see Chapter 4, *Floating-Point Numbers*), the parallel sum may not be the same as the serial sum. To perform a summation in parallel, you must be willing to tolerate these slight variations in your results.

5.6 Closing Notes[15]

5.6.1 Closing Notes

As they drop in price, multiprocessor systems are becoming far more common. These systems have many attractive features, including good price/performance, compatibility with workstations, large memories, high throughput, large shared memories, fast I/O, and many others. While these systems are strong in multiprogrammed server roles, they are also an affordable high performance computing resource for many organizations. Their cache-coherent shared-memory model allows multithreaded applications to be easily developed.

We have also examined some of the software paradigms that must be used to develop multithreaded applications. While you hopefully will never have to write C code with explicit threads like the examples in this chapter, it is nice to understand the fundamental operations at work on these multiprocessor systems. Using the FORTRAN language with an automatic parallelizing compiler, we have the advantage that these and many more details are left to the FORTRAN compiler and runtime library. At some point, especially on the most advanced architectures, you may have to explicitly program a multithreaded program using the types of techniques shown in this chapter.

One trend that has been predicted for some time is that we will begin to see multiple cache-coherent CPUs on a single chip once the ability to increase the clock rate on a single chip slows down. Imagine that your new $2000 workstation has four 1-GHz processors on a single chip. Sounds like a good time to learn how to write multithreaded programs!

5.7 Exercises[16]

5.7.1 Exercises

Exercise 5.1
Experiment with the fork code in this chapter. Run the program multiple times and see how the order of the messages changes. Explain the results.

Exercise 5.2
Experiment with the `create1` and `create3` codes in this chapter. Remove all of the `sleep( )` calls. Execute the programs several times on single and multiprocessor systems. Can you explain why the output changes from run to run in some situations and doesn't change in others?

Exercise 5.3
Experiment with the parallel sum code in this chapter. In the `SumFunc( )` routine, change the for-loop to:

```
for(i=start;i<end;i++ ) GlobSum = GlobSum + array[i];
```

[14] It is important to match the number of runnable threads to the available resources. In compute code, when there are more threads than available processors, the threads compete among themselves, causing unnecessary overhead and reducing the efficiency of your computation. See Appendix D for more details.

[15] This content is available online at <http://cnx.org/content/m32807/1.1/>.

[16] This content is available online at <http://cnx.org/content/m32810/1.1/>.

Remove the three lines at the end that get the mutex and update the `GlobSum`. Execute the code. Explain the difference in values that you see for `GlobSum`. Are the patterns different on a single processor and a multiprocessor? Explain the performance impact on a single processor and a multiprocessor.

Exercise 5.4

Explain how the following code segment could cause deadlock — two or more processes waiting for a resource that can't be relinquished:

```
...
call lock (lword1)
call lock (lword2)
...
call unlock (lword1)
call unlock (lword2)
.
.
.
call lock (lword2)
call lock (lword1)
   ...
call unlock (lword2)
call unlock (lword1)
   ...
```

Exercise 5.5

If you were to code the functionality of a spin-lock in C, it might look like this:

```
while (!lockword);
lockword = !lockword;
```

As you know from the first sections of the book, the same statements would be compiled into explicit loads and stores, a comparison, and a branch. There's a danger that two processes could each load `lockword`, find it unset, and continue on as if they owned the lock (we have a race condition). This suggests that spin-locks are implemented differently — that they're not merely the two lines of C above. How do you suppose they are implemented?

Chapter 6

Programming Shared-Memory Multiprocessors

6.1 Introduction[1]

6.1.1 Programming Shared-Memory Multiprocessors

In Chapter 10, Shared-Memory Multiprocessors, we examined the hardware used to implement shared-memory parallel processors and the software environment for a programmer who is using threads explicitly. In this chapter, we view these processors from a simpler vantage point. When programming these systems in FORTRAN, you have the advantage of the compiler's support of these systems. At the top end of ease of use, we can simply add a flag or two on the compilation of our well-written code, set an environment variable, and voilá, we are executing in parallel. If you want some more control, you can add directives to particular loops where you know better than the compiler how the loop should be executed.[2] First we examine how well-written loops can benefit from automatic parallelism. Then we will look at the types of directives you can add to your program to assist the compiler in generating parallel code. While this chapter refers to running your code in parallel, most of the techniques apply to the vector-processor supercomputers as well.

6.2 Automatic Parallelization[3]

6.2.1 Automatic Parallelization

So far in the book, we've covered the tough things you need to know to do parallel processing. At this point, assuming that your loops are clean, they use unit stride, and the iterations can all be done in parallel, all you have to do is turn on a compiler flag and buy a good parallel processor. For example, look at the following code:

```
PARAMETER(NITER=300,N=1000000)
```

[1] This content is available online at <http://cnx.org/content/m32812/1.1/>.

[2] If you have skipped all the other chapters in the book and jumped to this one, don't be surprised if some of the terminology is unfamiliar. While all those chapters seemed to contain endless boring detail, they did contain some basic terminology. So those of us who read all those chapters have some common terminology needed for this chapter. If you don't go back and read all the chapters, don't complain about the big words we keep using in this chapter!

[3] This content is available online at <http://cnx.org/content/m32821/1.1/>.

```
REAL*8 A(N),X(N),B(N),C
DO ITIME=1,NITER
  DO I=1,N
      A(I) = X(I) + B(I) * C
  ENDDO
  CALL WHATEVER(A,X,B,C)
ENDDO
```

Here we have an iterative code that satisfies all the criteria for a good parallel loop. On a good parallel processor with a modern compiler, you are two flags away from executing in parallel. On Sun Solaris systems, the `autopar` flag turns on the automatic parallelization, and the `loopinfo` flag causes the compiler to describe the particular optimization performed for each loop. To compile this code under Solaris, you simply add these flags to your `f77` call:

```
E6000: f77 -O3 -autopar -loopinfo -o daxpy daxpy.f
daxpy.f:
"daxpy.f", line 6: not parallelized, call may be unsafe
"daxpy.f", line 8: PARALLELIZED
E6000: /bin/time daxpy

real       30.9
user       30.7
sys         0.1
E6000:
```

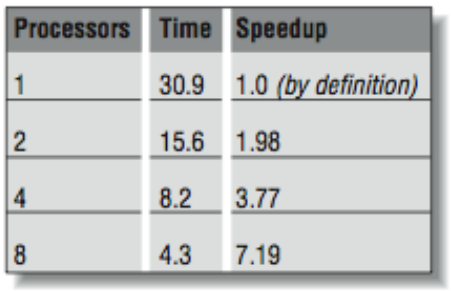
If you simply run the code, it's executed using one thread. However, the code is enabled for parallel processing for those loops that can be executed in parallel. To execute the code in parallel, you need to set the UNIX environment to the number of parallel threads you wish to use to execute the code. On Solaris, this is done using the `PARALLEL` variable:

```
E6000: setenv PARALLEL 1
E6000: /bin/time daxpy

real        30.9
user        30.7
sys          0.1
E6000: setenv PARALLEL 2
E6000: /bin/time daxpy

real        15.6
user        31.0
sys          0.2
E6000: setenv PARALLEL 4
E6000: /bin/time daxpy
```

```
real        8.2
user       32.0
sys         0.5
E6000: setenv PARALLEL 8
E6000: /bin/time daxpy

real        4.3
user       33.0
sys         0.8
```

Speedup is the term used to capture how much faster the job runs using N processors compared to the performance on one processor. It is computed by dividing the single processor time by the multiprocessor time for each number of processors. Figure 6.1 (Figure 11-1: Improving performance by adding processors) shows the wall time and speedup for this application.

Figure 11-1: Improving performance by adding processors



Processors	Time	Speedup
1	30.9	1.0 *(by definition)*
2	15.6	1.98
4	8.2	3.77
8	4.3	7.19

Figure 6.1

Figure 6.2 (Figure 11-2: Ideal and actual performance improvement) shows this information graphically, plotting speedup versus the number of processors.

Figure 11-2: Ideal and actual performance improvement

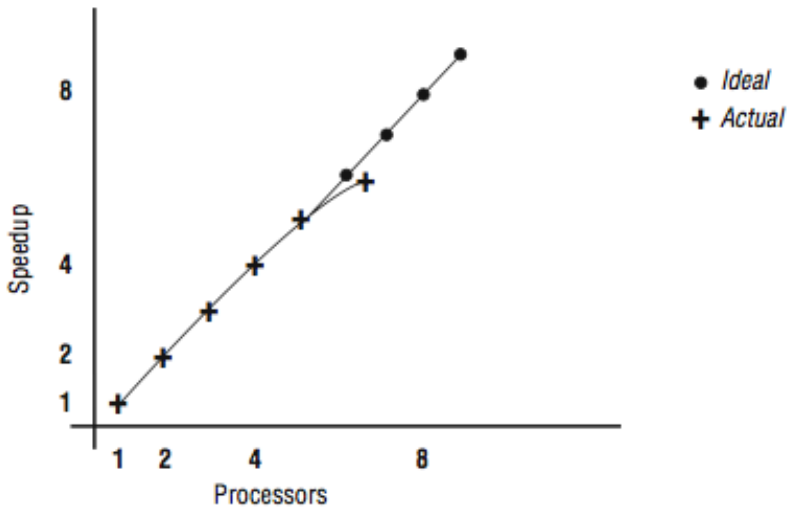


Figure 6.2

Note that for a while we get nearly perfect speedup, but we begin to see a measurable drop in speedup at four and eight processors. There are several causes for this. In all parallel applications, there is some portion of the code that can't run in parallel. During those nonparallel times, the other processors are waiting for work and aren't contributing to efficiency. This nonparallel code begins to affect the overall performance as more processors are added to the application.

So you say, "this is more like it!" and immediately try to run with 12 and 16 threads. Now, we see the graph in Figure 6.4 (Figure 11-4: Diminishing returns) and the data from Figure 6.3 (Figure 11-3: Increasing the number of threads).

Figure 11-3: Increasing the number of threads

Processors	Time	Speedup
1	30.9	1.0 *(by definition)*
2	15.6	1.98
4	8.2	3.77
8	4.3	7.19
12	14.2	2.18
16	57.0	0.57

Figure 6.3

Figure 11-4: Diminishing returns

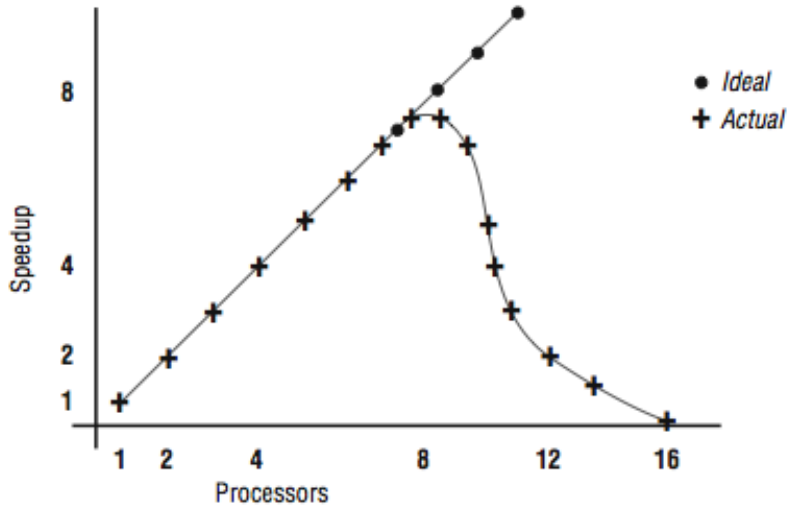


Figure 6.4

What has happened here? Things were going so well, and then they slowed down. We are running this program on a 16-processor system, and there are eight other active threads, as indicated below:

```
E6000:uptime
  4:00pm  up 19 day(s), 37 min(s), 5 users, load average: 8.00, 8.05, 8.14
E6000:
```

Once we pass eight threads, there are no available processors for our threads. So the threads must be time-shared between the processors, significantly slowing the overall operation. By the end, we are executing 16 threads on eight processors, and our performance is slower than with one thread. So it is important that you don't create too many threads in these types of applications.[4]

6.2.1.1 Compiler Considerations

Improving performance by turning on automatic parallelization is an example of the "smarter compiler" we discussed in earlier chapters. The addition of a single compiler flag has triggered a great deal of analysis on the part of the compiler including:

- Which loops can execute in parallel, producing the exact same results as the sequential executions of the loops? This is done by checking for dependencies that span iterations. A loop with no interiteration dependencies is called a `DOALL` loop.
- Which loops are worth executing in parallel? Generally very short loops gain no benefit and may execute more slowly when executing in parallel. As with loop unrolling, parallelism always has a cost. It is best used when the benefit far outweighs the cost.
- In a loop nest, which loop is the best candidate to be parallelized? Generally the best performance occurs when we parallelize the outermost loop of a loop nest. This way the overhead associated with beginning a parallel loop is amortized over a longer parallel loop duration.
- Can and should the loop nest be interchanged? The compiler may detect that the loops in a nest can be done in any order. One order may work very well for parallel code while giving poor memory performance. Another order may give unit stride but perform poorly with multiple threads. The compiler must analyze the cost/benefit of each approach and make the best choice.
- How do we break up the iterations among the threads executing a parallel loop? Are the iterations short with uniform duration, or long with wide variation of execution time? We will see that there are a number of different ways to accomplish this. When the programmer has given no guidance, the compiler must make an educated guess.

Even though it seems complicated, the compiler can do a surprisingly good job on a wide variety of codes. It is not magic, however. For example, in the following code we have a loop-carried flow dependency:

```
PROGRAM DEP
PARAMETER(NITER=300,N=1000000)
REAL*4 A(N)
```

[4] In Appendix D, How FORTRAN Manages Threads at Runtime, when we look at how the FORTRAN runtime library operates on these systems it will be much clearer why having more threads than avail- able processors has such a negative impact on performance.

```
DO ITIME=1,NITER
  CALL WHATEVER(A)
  DO I=2,N
      A(I) = A(I-1) + A(I) * C
  ENDDO
ENDDO
END
```

When we compile the code, the compiler gives us the following message:
dep.f:

```
E6000: f77 -O3 -autopar -loopinfo -o dep dep.f
dep.f:
"dep.f", line 6: not parallelized, call may be unsafe
"dep.f", line 8: not parallelized, unsafe dependence (a)
E6000:
```

The compiler throws its hands up in despair, and lets you know that the loop at Line 8 had an unsafe dependence, and so it won't automatically parallelize the loop. When the code is executed below, adding a thread does not affect the execution performance:

```
E6000:setenv PARALLEL 1
E6000:/bin/time dep

real       18.1
user       18.1
sys         0.0
E6000:setenv PARALLEL 2
E6000:/bin/time dep

real       18.3
user       18.2
sys         0.0
E6000:
```

A typical application has many loops. Not all the loops are executed in parallel. It's a good idea to run a profile of your application, and in the routines that use most of the CPU time, check to find out which loops are not being parallelized. Within a loop nest, the compiler generally chooses only one loop to execute in parallel.

6.2.1.2 Other Compiler Flags

In addition to the flags shown above, you may have other compiler flags available to you that apply across the entire program:

- You may have a compiler flag to enable the automatic parallelization of reduction operations. Because the order of additions can affect the final value when computing a sum of floating-point numbers, the compiler needs permission to parallelize summation loops.
- Flags that relax the compliance with IEEE floating-point rules may also give the compiler more flexibility when trying to parallelize a loop. However, you must be sure that it's not causing accuracy problems in other areas of your code.
- Often a compiler has a flag called "unsafe optimization" or "assume no dependencies." While this flag may indeed enhance the performance of an application with loops that have dependencies, it almost certainly produces incorrect results.

There is some value in experimenting with a compiler to see the particular combination that will yield good performance across a variety of applications. Then that set of compiler options can be used as a starting point when you encounter a new application.

6.3 Assisting the Compiler[5]

6.3.1 Assisting the Compiler

If it were all that simple, you wouldn't need this book. While compilers are extremely clever, there is still a lot of ways to improve the performance of your code without sacrificing its portability. Instead of converting the whole program to C and using a thread library, you can assist the compiler by adding compiler directives to our source code.

Compiler directives are typically inserted in the form of stylized FORTRAN comments. This is done so that a nonparallelizing compiler can ignore them and just look at the FORTRAN code, sans comments. This allows to you tune your code for parallel architectures without letting it run badly on a wide range of single-processor systems.

There are two categories of parallel-processing comments:

- Assertions
- Manual parallelization directives

Assertions tell the compiler certain things that you as the programmer know about the code that it might not guess by looking at the code. Through the assertions, you are attempting to assuage the compiler's doubts about whether or not the loop is eligible for parallelization. When you use directives, you are taking full responsibility for the correct execution of the program. You are telling the compiler what to parallelize and how to do it. You take full responsibility for the output of the program. If the program produces meaningless results, you have no one to blame but yourself.

6.3.1.1 Assertions

In a previous example, we compiled a program and received the following output:

```
E6000: f77 -O3 -autopar -loopinfo -o dep dep.f
dep.f:
"dep.f", line 6: not parallelized, call may be unsafe
"dep.f", line 8: not parallelized, unsafe dependence (a)
E6000:
```

[5]This content is available online at <http://cnx.org/content/m32814/1.1/>.

An uneducated programmer who has not read this book (or has not looked at the code) might exclaim, "What unsafe dependence, I never put one of those in my code!" and quickly add a *no dependencies* assertion. This is the essence of an assertion. Instead of telling the compiler to simply parallelize the loop, the programmer is telling the compiler that its conclusion that there is a dependence is incorrect. Usually the net result is that the compiler does indeed parallelize the loop.

We will briefly review the types of assertions that are typically supported by these compilers. An assertion is generally added to the code using a stylized comment.

6.3.1.1.1 No dependencies

A **no dependencies** or **ignore dependencies** directive tells the compiler that references don't overlap. That is, it tells the compiler to generate code that may execute incorrectly if there **are** dependencies. You're saying, "I know what I'm doing; it's OK to overlap references." A no dependencies directive might help the following loop:

```
DO I=1,N
  A(I) = A(I+K) * B(I)
ENDDO
```

If you know that `k` is greater than `-1` or less than `-n`, you can get the compiler to parallelize the loop:

```
C$ASSERT NO_DEPENDENCIES
     DO I=1,N
       A(I) = A(I+K) * B(I)
     ENDDO
```

Of course, blindly telling the compiler that there are no dependencies is a prescription for disaster. If `k` equals `-1`, the example above becomes a recursive loop.

6.3.1.1.2 Relations

You will often see loops that contain some potential dependencies, making them bad candidates for a no dependencies directive. However, you may be able to supply some local facts about certain variables. This allows partial parallelization without compromising the results. In the code below, there are two potential dependencies because of subscripts involving k and j:

```
for (i=0; i<n; i++) {
    a[i] = a[i+k] * b[i];
    c[i] = c[i+j] * b[i];
}
```

Perhaps we know that there are no conflicts with references to `a[i]` and `a[i+k]`. But maybe we aren't so sure about `c[i]` and `c[i+j]`. Therefore, we can't say in general that there are no dependencies. However, we may be able to say something explicit about `k` (like "`k` is always greater than `-1`"), leaving `j` out of it. This information about the relationship of one expression to another is called a **relation assertion**. Applying a relation assertion allows the compiler to apply its optimization to the first statement in the loop, giving us partial parallelization.[6]

Again, if you supply inaccurate testimony that leads the compiler to make unsafe optimizations, your answer may be wrong.

6.3.1.1.3 Permutations

As we have seen elsewhere, when elements of an array are indirectly addressed, you have to worry about whether or not some of the subscripts may be repeated. In the code below, are the values of K(I) all unique? Or are there duplicates?

```
DO I=1,N
  A(K(I)) = A(K(I)) + B(I) * C
END DO
```

If you know there are no duplicates in `K` (i.e., that `A(K(I))` is a permutation), you can inform the compiler so that iterations can execute in parallel. You supply the information using a *permutation assertion*.

6.3.1.1.4 No equivalences

Equivalenced arrays in FORTRAN programs provide another challenge for the compiler. If any elements of two equivalenced arrays appear in the same loop, most compilers assume that references could point to the same memory storage location and optimize very conservatively. This may be true even if it is abundantly apparent to you that there is no overlap whatsoever.

You inform the compiler that references to equivalenced arrays are safe with a *no equivalences* assertion. Of course, if you don't use equivalences, this assertion has no effect.

6.3.1.1.5 Trip count

Each loop can be characterized by an average number of iterations. Some loops are never executed or go around just a few times. Others may go around hundreds of times:

```
C$ASSERT TRIPCOUNT>100
     DO I=L,N
       A(I) = B(I) + C(I)
     END DO
```

Your compiler is going to look at every loop as a candidate for unrolling or parallelization. It's working in the dark, however, because it can't tell which loops are important and tries to optimize them all. This can lead to the surprising experience of seeing your runtime go up after optimization!

[6] Notice that, if you were tuning by hand, you could split this loop into two: one parallelizable and one not.

A *trip count assertion* provides a clue to the compiler that helps it decide how much to unroll a loop or when to parallelize a loop.[7] Loops that aren't important can be identified with low or zero trip counts. Important loops have high trip counts.

6.3.1.1.6 Inline substitution

If your compiler supports procedure inlining, you can use directives and command-line switches to specify how many nested levels of procedures you would like to inline, thresholds for procedure size, etc. The vendor will have chosen reasonable defaults.

Assertions also let you choose subroutines that you think are good candidates for inlining. However, subject to its thresholds, the compiler may reject your choices. Inlining could expand the code so much that increased memory activity would claim back gains made by eliminating the procedure call. At higher optimization levels, the compiler is often capable of making its own choices for inlining candidates, provided it can find the source code for the routine under consideration.

Some compilers support a feature called *interprocedural analysis*. When this is done, the compiler looks across routine boundaries for its data flow analysis. It can perform significant optimizations across routine boundaries, including automatic inlining, constant propagation, and others.

6.3.1.1.7 No side effects

Without interprocedural analysis, when looking at a loop, if there is a subroutine call in the middle of the loop, the compiler has to treat the subroutine as if it will have the worst possible side effects. Also, it has to assume that there are dependencies that prevent the routine from executing simultaneously in two different threads.

Many routines (especially functions) don't have any side effects and can execute quite nicely in separate threads because each thread has its own private call stack and local variables. If the routine is meaty, there will be a great deal of benefit in executing it in parallel.

Your computer may allow you to add a directive that tells you if successive sub-routine calls are independent:

```
C$ASSERT NO_SIDE_EFFECTS
     DO I=1,N
       CALL BIGSTUFF (A,B,C,I,J,K)
     END DO
```

Even if the compiler has all the source code, use of common variables or equivalences may mask call independence.

6.3.1.2 Manual Parallelism

At some point, you get tired of giving the compiler advice and hoping that it will reach the conclusion to parallelize your loop. At that point you move into the realm of manual parallelism. Luckily the programming model provided in FORTRAN insulates you from much of the details of exactly how multiple threads are managed at runtime. You generally control explicit parallelism by adding specially formatted comment lines to your source code. There are a wide variety of formats of these directives. In this section, we use the syntax that is part of the OpenMP (see [8]) standard. You generally find similar capabilities in each of the

[7] The assertion is made either by hand or from a profiler.

[8] http://cnx.org/content/m32814/latest/www.openmp.org

vendor compilers. The precise syntax varies slightly from vendor to vendor. (That alone is a good reason to have a standard.)

The basic programming model is that you are executing a section of code with either a single thread or multiple threads. The programmer adds a directive to summon additional threads at various points in the code. The most basic construct is called the *parallel region*.

6.3.1.2.1 Parallel regions

In a parallel region, the threads simply appear between two statements of straight-line code. A very trivial example might be the following using the OpenMP directive syntax:

```
        PROGRAM ONE
        EXTERNAL OMP_GET_THREAD_NUM, OMP_GET_MAX_THREADS
        INTEGER OMP_GET_THREAD_NUM, OMP_GET_MAX_THREADS
        IGLOB = OMP_GET_MAX_THREADS()
        PRINT *,'Hello There'
C$OMP PARALLEL PRIVATE(IAM), SHARED(IGLOB)
        IAM = OMP_GET_THREAD_NUM()
        PRINT *, 'I am ', IAM, ' of ', IGLOB
C$OMP END PARALLEL
        PRINT *,'All Done'
        END
```

The `C$OMP` is the sentinel that indicates that this is a directive and not just another comment. The output of the program when run looks as follows:

```
% setenv OMP_NUM_THREADS 4
% a.out
Hello There
I am 0 of 4
I am 3 of 4
I am 1 of 4
I am 2 of 4
All Done
%
```

Execution begins with a single thread. As the program encounters the `PARALLEL` directive, the other threads are activated to join the computation. So in a sense, as execution passes the first directive, one thread becomes four. Four threads execute the two statements between the directives. As the threads are executing independently, the order in which the print statements are displayed is somewhat random. The threads wait at the `END PARALLEL` directive until all threads have arrived. Once all threads have completed the parallel region, a single thread continues executing the remainder of the program.

In Figure 6.5 (Figure 11-5: data interactions during a parallel region), the `PRIVATE(IAM)` indicates that the `IAM` variable is not shared across all the threads but instead, each thread has its own private version of the variable. The `IGLOB` variable is shared across all the threads. Any modification of `IGLOB` appears in all the other threads instantly, within the limitations of the cache coherency.

Figure 11-5: data interactions during a parallel region

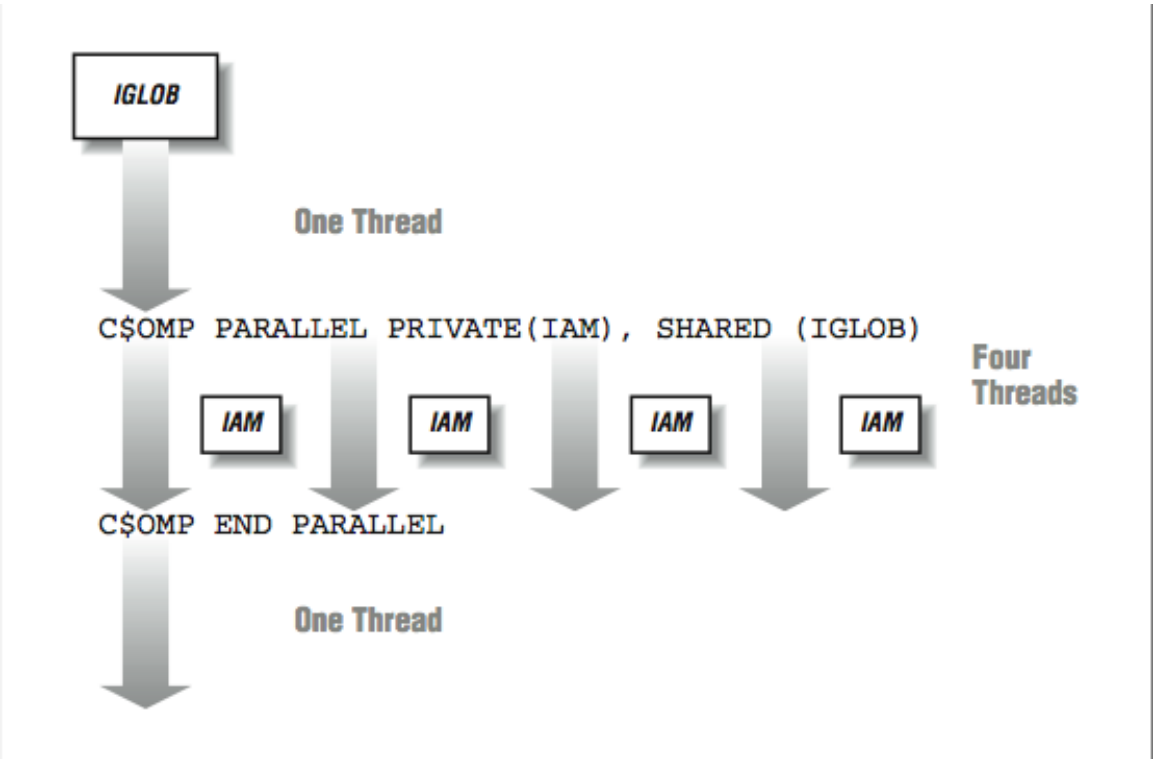


Figure 6.5

During the parallel region, the programmer typically divides the work among the threads. This pattern of going from single-threaded to multithreaded execution may be repeated many times throughout the execution of an application.

Because input and output are generally not thread-safe, to be completely correct, we should indicate that the print statement in the parallel section is only to be executed on one processor at any one time. We use a directive to indicate that this section of code is a critical section. A lock or other synchronization mechanism ensures that no more than one processor is executing the statements in the critical section at any one time:

```
C$OMP CRITICAL
        PRINT *, 'I am ', IAM, ' of ', IGLOB
C$OMP END CRITICAL
```

6.3.1.2.2 Parallel loops

Quite often the areas of the code that are most valuable to execute in parallel are loops. Consider the following loop:

```
DO I=1,1000000
  TMP1 = ( A(I) ** 2 ) + ( B(I) ** 2 )
  TMP2 = SQRT(TMP1)
  B(I) = TMP2
ENDDO
```

To manually parallelize this loop, we insert a directive at the beginning of the loop:

```
C$OMP PARALLEL DO
      DO I=1,1000000
        TMP1 = ( A(I) ** 2 ) + ( B(I) ** 2 )
        TMP2 = SQRT(TMP1)
        B(I) = TMP2
      ENDDO
C$OMP END PARALLEL DO
```

When this statement is encountered at runtime, the single thread again summons the other threads to join the computation. However, before the threads can start working on the loop, there are a few details that must be handled. The `PARALLEL DO` directive accepts the data classification and scoping clauses as in the parallel section directive earlier. We must indicate which variables are shared across all threads and which variables have a separate copy in each thread. It would be a disaster to have `TMP1` and `TMP2` shared across threads. As one thread takes the square root of `TMP1`, another thread would be resetting the contents of `TMP1`. `A(I)` and `B(I)` come from outside the loop, so they must be shared. We need to augment the directive as follows:

```
C$OMP PARALLEL DO SHARED(A,B) PRIVATE(I,TMP1,TMP2)
      DO I=1,1000000
        TMP1 = ( A(I) ** 2 ) + ( B(I) ** 2 )
        TMP2 = SQRT(TMP1)
        B(I) = TMP2
      ENDDO
C$OMP END PARALLEL DO
```

The iteration variable I also must be a thread-private variable. As the different threads increment their way through their particular subset of the arrays, they don't want to be modifying a global value for I.

There are a number of other options as to how data will be operated on across the threads. This summarizes some of the other data semantics available:

Firstprivate: These are thread-private variables that take an initial value from the global variable of the same name immediately before the loop begins executing.

Lastprivate: These are thread-private variables except that the thread that executes the last iteration of the loop copies its value back into the global variable of the same name.

Reduction: This indicates that a variable participates in a reduction operation that can be safely done in parallel. This is done by forming a partial reduction using a local variable in each thread and then combining the partial results at the end of the loop.

Each vendor may have different terms to indicate these data semantics, but most support all of these common semantics. Figure 6.6 (Figure 11-6: Variables during a parallel region) shows how the different types of data semantics operate.

Now that we have the data environment set up for the loop, the only remaining problem that must be solved is which threads will perform which iterations. It turns out that this is not a trivial task, and a wrong choice can have a significant negative impact on our overall performance.

6.3.1.2.3 Iteration scheduling

There are two basic techniques (along with a few variations) for dividing the iterations in a loop between threads. We can look at two extreme examples to get an idea of how this works:

```
C VECTOR ADD
      DO IPROB=1,10000
         A(IPROB) = B(IPROB) + C(IPROB)
      ENDDO

C PARTICLE TRACKING
       DO IPROB=1,10000
          RANVAL = RAND(IPROB)
          CALL ITERATE_ENERGY(RANVAL) ENDDO
       ENDDO
```

Figure 11-6: Variables during a parallel region

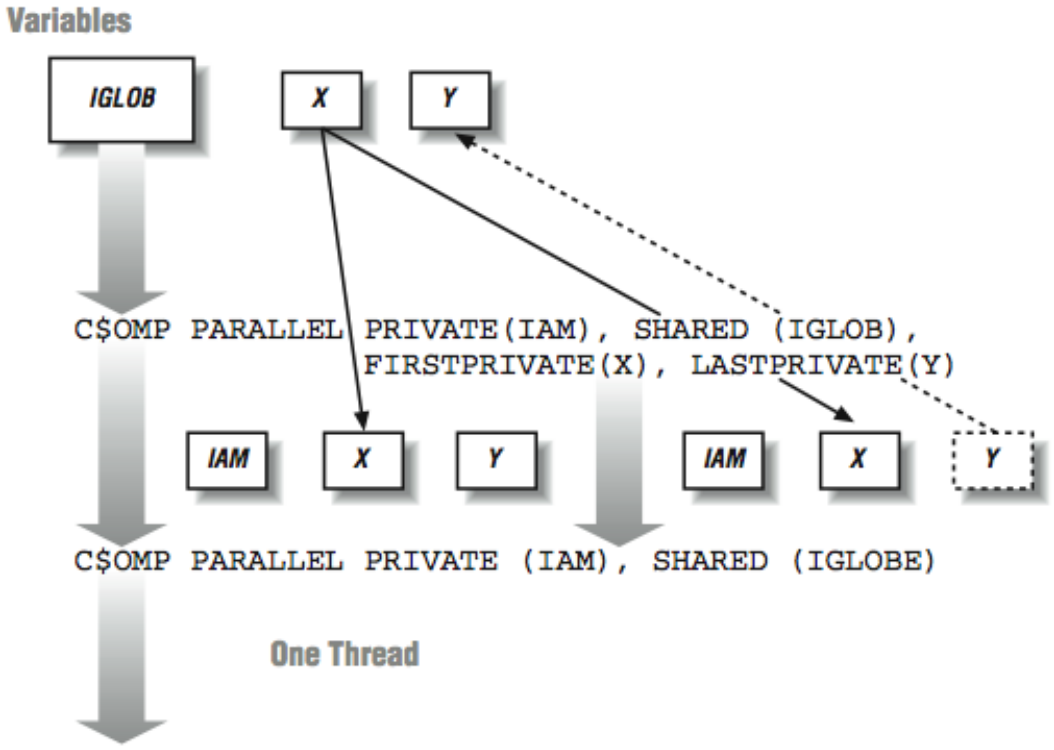


Figure 6.6

In both loops, all the computations are independent, so if there were 10,000 processors, each processor could execute a single iteration. In the vector-add example, each iteration would be relatively short, and the execution time would be relatively constant from iteration to iteration. In the particle tracking example, each iteration chooses a random number for an initial particle position and iterates to find the minimum energy. Each iteration takes a relatively long time to complete, and there will be a wide variation of completion times from iteration to iteration.

These two examples are effectively the ends of a continuous spectrum of the iteration scheduling challenges facing the FORTRAN parallel runtime environment:

Static

At the beginning of a parallel loop, each thread takes a fixed continuous portion of iterations of the loop based on the number of threads executing the loop.

Dynamic

With dynamic scheduling, each thread processes a chunk of data and when it has completed processing, a new chunk is processed. The chunk size can be varied by the programmer, but is fixed for the duration of the loop.

These two example loops can show how these iteration scheduling approaches might operate when ex-

ecuting with four threads. In the vector-add loop, static scheduling would distribute iterations 1–2500 to Thread 0, 2501–5000 to Thread 1, 5001–7500 to Thread 2, and 7501–10000 to Thread 3. In Figure 6.7 (Figure 11-7: Iteration assignment for static scheduling), the mapping of iterations to threads is shown for the static scheduling option.

Figure 11-7: Iteration assignment for static scheduling

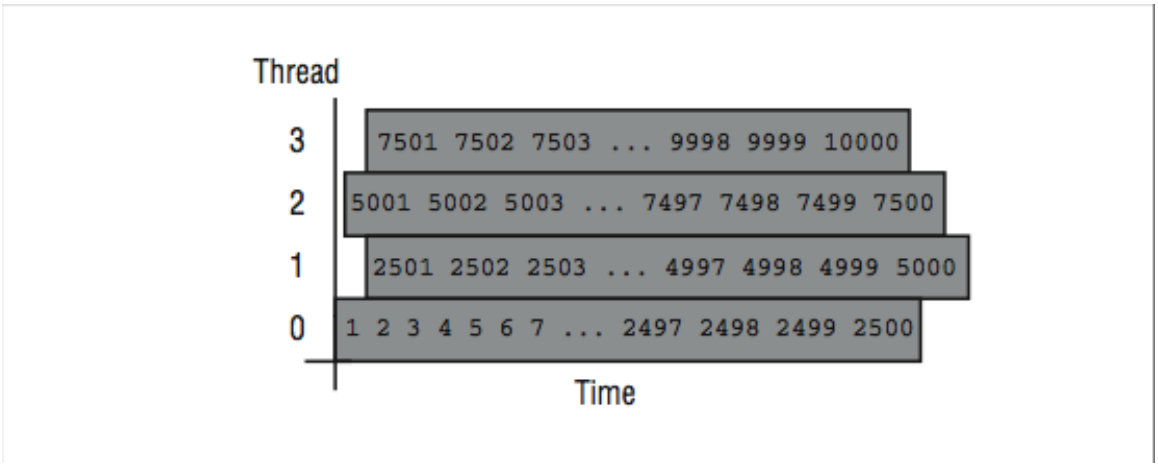


Figure 6.7

Since the loop body (a single statement) is short with a consistent execution time, static scheduling should result in roughly the same amount of overall work (and time if you assume a dedicated CPU for each thread) assigned to each thread per loop execution.

An advantage of static scheduling may occur if the entire loop is executed repeatedly. If the same iterations are assigned to the same threads that happen to be running on the same processors, the cache might actually contain the values for `A`, `B`, and `C` from the previous loop execution.[9] The runtime pseudo-code for static scheduling in the first loop might look as follows:

```
C VECTOR ADD - Static Scheduled
      ISTART = (THREAD_NUMBER * 2500 ) + 1
      IEND = ISTART + 2499
      DO ILOCAL = ISTART,IEND
        A(ILOCAL) = B(ILOCAL) + C(ILOCAL)
      ENDDO
```

It's not always a good strategy to use the static approach of giving a fixed number of iterations to each thread. If this is used in the second loop example, long and varying iteration times would result in poor load

[9] The operating system and runtime library actually go to some lengths to try to make this happen. This is another reason not to have more threads than available processors, which causes unnecessary context switching.

balancing. A better approach is to have each processor simply get the next value for `IPROB` each time at the top of the loop.

That approach is called *dynamic scheduling*, and it can adapt to widely varying iteration times. In Figure 6.8 (Figure 11-8: Iteration assignment in dynamic scheduling), the mapping of iterations to processors using dynamic scheduling is shown. As soon as a processor finishes one iteration, it processes the next available iteration in order.

Figure 11-8: Iteration assignment in dynamic scheduling

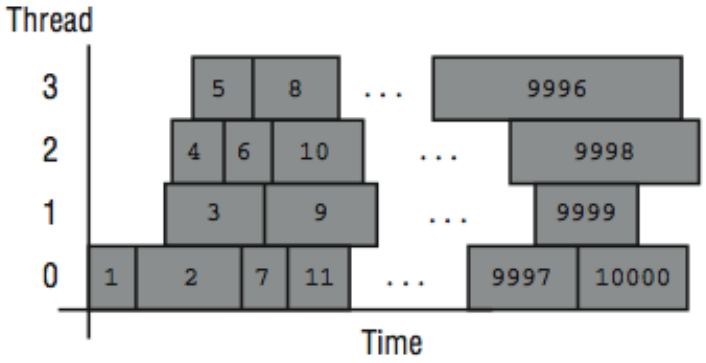


Figure 6.8

If a loop is executed repeatedly, the assignment of iterations to threads may vary due to subtle timing issues that affect threads. The pseudo-code for the dynamic scheduled loop at runtime is as follows:

```
C PARTICLE TRACKING - Dynamic Scheduled
      IPROB = 0
      WHILE (IPROB <= 10000 )
        BEGIN_CRITICAL_SECTION
          IPROB = IPROB + 1
          ILOCAL = IPROB
        END_CRITICAL_SECTION
        RANVAL = RAND(ILOCAL)
        CALL ITERATE_ENERGY(RANVAL)
      ENDWHILE
```

`ILOCAL` is used so that each thread knows which iteration is currently processing. The `IPROB` value is altered by the next thread executing the critical section.

While the dynamic iteration scheduling approach works well for this particular loop, there is a significant negative performance impact if the programmer were to use the wrong approach for a loop. For example, if the dynamic approach were used for the vector-add loop, the time to process the critical section to determine which iteration to process may be larger than the time to actually process the iteration. Furthermore, any

cache affinity of the data would be effectively lost because of the virtually random assignment of iterations to processors.

In between these two approaches are a wide variety of techniques that operate on a chunk of iterations. In some techniques the chunk size is fixed, and in others it varies during the execution of the loop. In this approach, a chunk of iterations are grabbed each time the critical section is executed. This reduces the scheduling overhead, but can have problems in producing a balanced execution time for each processor. The runtime is modified as follows to perform the particle tracking loop example using a chunk size of 100:

```
IPROB = 1
CHUNKSIZE = 100
WHILE (IPROB <= 10000 )
  BEGIN_CRITICAL_SECTION
    ISTART = IPROB
    IPROB = IPROB + CHUNKSIZE
  END_CRITICAL_SECTION
  DO ILOCAL = ISTART,ISTART+CHUNKSIZE-1
    RANVAL = RAND(ILOCAL)
    CALL ITERATE_ENERGY(RANVAL)
  ENDDO
ENDWHILE
```

The choice of chunk size is a compromise between overhead and termination imbalance. Typically the programmer must get involved through directives in order to control chunk size.

Part of the challenge of iteration distribution is to balance the cost (or existence) of the critical section against the amount of work done per invocation of the critical section. In the ideal world, the critical section would be free, and all scheduling would be done dynamically. Parallel/vector supercomputers with hardware assistance for load balancing can nearly achieve the ideal using dynamic approaches with relatively small chunk size.

Because the choice of loop iteration approach is so important, the compiler relies on directives from the programmer to specify which approach to use. The following example shows how we can request the proper iteration scheduling for our loops:

```
C VECTOR ADD
C$OMP PARALLEL DO PRIVATE(IPROB) SHARED(A,B,C) SCHEDULE(STATIC)
      DO IPROB=1,10000
         A(IPROB) = B(IPROB) + C(IPROB)
      ENDDO
C$OMP END PARALLEL DO
C PARTICLE TRACKING
C$OMP PARALLEL DO PRIVATE(IPROB,RANVAL) SCHEDULE(DYNAMIC)
      DO IPROB=1,10000
         RANVAL = RAND(IPROB)
         CALL ITERATE_ENERGY(RANVAL)
      ENDDO
C$OMP END PARALLEL DO
```

6.4 Closing Notes[10]

6.4.1 Closing Notes

Using data flow analysis and other techniques, modern compilers can peer through the clutter that we programmers innocently put into our code and see the patterns of the actual computations. In the field of high performance computing, having great parallel hardware and a lousy automatic parallelizing compiler generally results in no sales. Too many of the benchmark rules allow only a few compiler options to be set.

Physicists and chemists are interested in physics and chemistry, not computer science. If it takes 1 hour to execute a chemistry code without modifications and after six weeks of modifications the same code executes in 20 minutes, which is better? Well from a chemist's point of view, one took an hour, and the other took 1008 hours and 20 minutes, so the answer is obvious.[11] Although if the program were going to be executed thousands of times, the tuning might be a win for the programmer. The answer is even more obvious if it again takes six weeks to tune the program every time you make a modification to the program.

In some ways, assertions have become less popular than directives. This is due to two factors: (1) compilers are getting better at detecting parallelism even if they have to rewrite some code to do so, and (2) there are two kinds of programmers: those who know exactly how to parallelize their codes and those who turn on the "safe" auto-parallelize flags on their codes. Assertions fall in the middle ground, somewhere between where the programmer does not want to control all the details but kind of feels that the loop can be parallelized.

You can get online documentation of the OpenMP syntax used in these examples at www.openmp.org[12] .

6.5 Exercises[13]

6.5.1 Exercises

Exercise 6.1
Take a static, highly parallel program with a relative large inner loop. Compile the application for parallel execution. Execute the application increasing the threads. Examine the behavior when the number of threads exceed the available processors. See if different iteration scheduling approaches make a difference.

Exercise 6.2
Take the following loop and execute with several different iteration scheduling choices. For chunk-based scheduling, use a large chunk size, perhaps 100,000. See if any approach performs better than static scheduling:

```
DO I=1,4000000
  A(I) = B(I) * 2.34
ENDDO
```

Exercise 6.3
Execute the following loop for a range of values for N from 1 to 16 million:

[10] This content is available online at <http://cnx.org/content/m32820/1.1/>.
[11] On the other hand, if the person is a computer scientist, improving the performance might result in anything from a poster session at a conference to a journal article! This makes for lots of intra-departmental masters degree projects.
[12] http://cnx.org/content/m32820/latest/www.openmp.org
[13] This content is available online at <http://cnx.org/content/m32819/1.1/>.

```
      DO I=1,N
        A(I) = B(I) * 2.34
      ENDDO
```

Run the loop in a single processor. Then force the loop to run in parallel. At what point do you get better performance on multiple processors? Do the number of threads affect your observations?

Exercise 6.4

Use an explicit parallelization directive to execute the following loop in parallel with a chunk size of 1:

```
      J = 0
C$OMP PARALLEL DO PRIVATE(I) SHARED(J) SCHEDULE(DYNAMIC)
      DO I=1,1000000
       J = J + 1
      ENDDO
      PRINT *, J
C$OMP END PARALLEL DO
```

Execute the loop with a varying number of threads, including one. Also compile and execute the code in serial. Compare the output and execution times. What do the results tell you about cache coherency? About the cost of moving data from one cache to another, and about critical section costs?

Attributions

Collection: *High Performance Computing*
Edited by: Charles Severance
URL: http://cnx.org/content/col11136/1.2/
License: http://creativecommons.org/licenses/by/3.0/

Module: "1.0 Introduction to the Connexions Edition"
Used here as: "Introduction to the Connexions Edition"
By: Charles Severance
URL: http://cnx.org/content/m32709/1.1/
Pages: 1-2
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "1.1 Introduction to High Performance Computing"
Used here as: "Introduction to High Performance Computing"
By: Charles Severance
URL: http://cnx.org/content/m32676/1.1/
Pages: 2-4
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "3.1 Introduction"
Used here as: "Introduction"
By: Charles Severance
URL: http://cnx.org/content/m32733/1.1/
Pages: 5-6
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "3.2 Memory Technology"
Used here as: "Memory Technology"
By: Charles Severance
URL: http://cnx.org/content/m32716/1.1/
Pages: 6-7
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "3.3 Registers"
Used here as: "Registers"
By: Charles Severance
URL: http://cnx.org/content/m32681/1.1/
Page: 7
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "3.4 Caches"
Used here as: "Caches"
By: Charles Severance
URL: http://cnx.org/content/m32725/1.1/
Pages: 8-11
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "3.5 Cache Organization"
Used here as: "Cache Organization"
By: Charles Severance
URL: http://cnx.org/content/m32722/1.1/
Pages: 11-15

Module: "3.6 Virtual Memory"
Used here as: "Virtual Memory"
By: Charles Severance
URL: http://cnx.org/content/m32728/1.1/
Pages: 15-18

Module: "3.7 Improving Memory Performance"
Used here as: "Improving Memory Performance"
By: Charles Severance
URL: http://cnx.org/content/m32736/1.1/
Pages: 18-25

Module: "3.8 Closing Notes"
Used here as: "Closing Notes"
By: Charles Severance
URL: http://cnx.org/content/m32690/1.1/
Page: 26

Module: "3.9 Exercises"
Used here as: "Exercises"
By: Charles Severance
URL: http://cnx.org/content/m32698/1.1/
Pages: 26-27

Module: "4.1 Introduction"
Used here as: "Introduction"
By: Charles Severance
URL: http://cnx.org/content/m32739/1.1/
Page: 29

Module: "4.2 Reality"
Used here as: "Reality"
By: Charles Severance
URL: http://cnx.org/content/m32741/1.1/
Page: 29
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.3 Representation"
Used here as: "Representation"
By: Charles Severance
URL: http://cnx.org/content/m32772/1.1/
Pages: 30-33
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.4 Effects of Floating-Point Representation"
Used here as: "Effects of Floating-Point Representation"
By: Charles Severance
URL: http://cnx.org/content/m32755/1.1/
Pages: 33-34
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.5 More Algebra That Doesn't Work"
Used here as: "More Algebra That Doesn't Work"
By: Charles Severance
URL: http://cnx.org/content/m32754/1.1/
Pages: 34-36
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.6 Improving Accuracy Using Guard Digits"
Used here as: "Improving Accuracy Using Guard Digits"
By: Charles Severance
URL: http://cnx.org/content/m32744/1.1/
Page: 37
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.7 History of IEEE Floating-Point Format"
Used here as: "History of IEEE Floating-Point Format"
By: Charles Severance
URL: http://cnx.org/content/m32770/1.1/
Pages: 37-40
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.8 IEEE Operations"
Used here as: "IEEE Operations"
By: Charles Severance
URL: http://cnx.org/content/m32756/1.1/
Pages: 40-42
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.9 Special Values"
Used here as: "Special Values"
By: Charles Severance
URL: http://cnx.org/content/m32758/1.1/
Pages: 42-43
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.10 Exceptions and Traps"
Used here as: "Exceptions and Traps"
By: Charles Severance
URL: http://cnx.org/content/m32760/1.1/
Pages: 43-44
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.11 Compiler Issues"
Used here as: "Compiler Issues"
By: Charles Severance
URL: http://cnx.org/content/m32762/1.1/
Page: 44
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.12 Closing Notes"
Used here as: "Closing Notes"
By: Charles Severance
URL: http://cnx.org/content/m32768/1.1/
Page: 45
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "4.13 Exercises"
Used here as: "Exercises"
By: Charles Severance
URL: http://cnx.org/content/m32765/1.1/
Pages: 45-46
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "9.1 Introduction"
Used here as: "Introduction"
By: Charles Severance
URL: http://cnx.org/content/m32775/1.1/
Pages: 47-48
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "9.2 Dependencies"
Used here as: "Dependencies"
By: Charles Severance
URL: http://cnx.org/content/m32777/1.1/
Pages: 48-57
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "9.3 Loops"
Used here as: "Loops"
By: Charles Severance
URL: http://cnx.org/content/m32784/1.1/
Pages: 57-59
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "9.4 Loop-Carried Dependencies"
Used here as: "Loop-Carried Dependencies "
By: Charles Severance
URL: http://cnx.org/content/m32782/1.1/
Pages: 59-64
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "9.5 Ambiguous References"
Used here as: "Ambiguous References"
By: Charles Severance
URL: http://cnx.org/content/m32788/1.1/
Pages: 64-67
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "9.6 Closing Notes"
Used here as: "Closing Notes"
By: Charles Severance
URL: http://cnx.org/content/m32789/1.1/
Page: 67
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "9.7 Exercises"
Used here as: "Exercises"
By: Charles Severance
URL: http://cnx.org/content/m32792/1.1/
Pages: 67-69
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "10.1 Introduction"
Used here as: "Introduction"
By: Charles Severance
URL: http://cnx.org/content/m32797/1.1/
Pages: 71-72
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "10.2 Symmetric Multiprocessing Hardware"
Used here as: "Symmetric Multiprocessing Hardware"
By: Charles Severance
URL: http://cnx.org/content/m32794/1.1/
Pages: 72-77
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "10.3 Multiprocessor Software Concepts"
Used here as: "Multiprocessor Software Concepts "
By: Charles Severance
URL: http://cnx.org/content/m32800/1.1/
Pages: 77-89
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "10.4 Techniques for Multithreaded Programs"
Used here as: "Techniques for Multithreaded Programs"
By: Charles Severance
URL: http://cnx.org/content/m32802/1.1/
Pages: 89-92
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "10.5. A Real Example"
Used here as: "A Real Example "
By: Charles Severance
URL: http://cnx.org/content/m32804/1.1/
Pages: 92-95
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "10.6 Closing Notes"
Used here as: "Closing Notes"
By: Charles Severance
URL: http://cnx.org/content/m32807/1.1/
Page: 95
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "10.7 Exercises"
Used here as: "Exercises"
By: Charles Severance
URL: http://cnx.org/content/m32810/1.1/
Pages: 95-96
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "11.1 Introduction"
Used here as: " Introduction"
By: Charles Severance
URL: http://cnx.org/content/m32812/1.1/
Page: 97
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "11.2 Automatic Parallelization"
Used here as: "Automatic Parallelization"
By: Charles Severance
URL: http://cnx.org/content/m32821/1.1/
Pages: 97-104
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "11.3 Assisting the Compiler"
Used here as: "Assisting the Compiler"
By: Charles Severance
URL: http://cnx.org/content/m32814/1.1/
Pages: 104-115
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "11.4 Closing Notes"
Used here as: "Closing Notes"
By: Charles Severance
URL: http://cnx.org/content/m32820/1.1/
Page: 116
Copyright: Charles Severance
License: http://creativecommons.org/licenses/by/3.0/

Module: "11.5 Exercises"
Used here as: "Exercises"
By: Charles Severance
URL: http://cnx.org/content/m32819/1.1/
Pages: 116-117

About Connexions

Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.