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

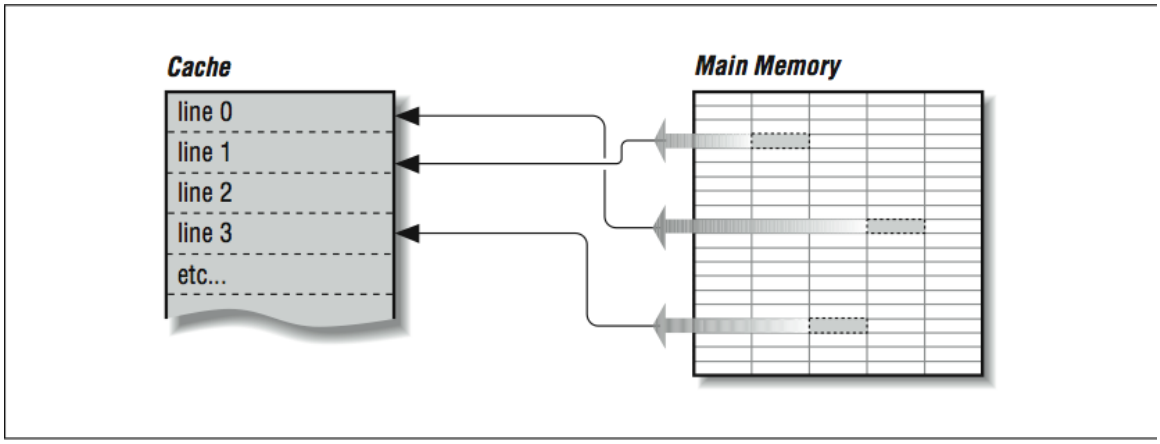

Figure 2.1

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/>.