# Chapter 2

# Performance Issues: General

Here is an all-too-common scenario: An analyst acquires a brand new multicore machine, capable of wondrous things. With great excitement, he codes up his favorite large problem on the new machine—only to find that the parallel version runs more slowly than the serial one. What a disappointment!

Though you are no doubt anxious to get to some more code, a firm grounding in the infrastructural issues will prove to be quite valuable indeed, hence the need for this chapter. So, let's see what factors can lead to such a situation in which our hapless analyst above sees his wonderful plans go awry.

## 2.1 Obstacles to Speed

Let's refer to the computational entities as *processes*, such as the workers in the case of **snow**. There are two main performance issues in parallel programming:

- *Communications overhead:* Typically data must be transferred back and forth between processes. This takes time, which can take quite a toll on performance.

  In addition, the processes can get in each other's way if they all try to access the same data at once. Or they can collide when trying to

access the same communications channel, the same memory module, and so on. This is another sap on speed.

The term *granularity* is used to refer, roughly, to the ratio of computation to overhead. *Coarse-grained* algorithms involve large enough chunks of computation that the overhead isn't much of a problem. In *fine-grained* algorithms, we really need to avoid overhead as much as possible.

- *Load balance:* As noted in the last chapter, if we are not careful in the way in which we assign work to processes, we risk assigning much more work to some than to others. This compromises performance, as it leaves some processes unproductive at the end of the run, while there is still work to be done.

There are a number of issues of this sort that arise generally enough to be collected into this chapter, as an "early warning" of issues that can arise. This is just an overview, with details coming in subsequent chapters, but being forewarned of the problems will make it easier to recognize them as they arise.

## 2.2 Performance and Hardware Structures

*Scorecards, scorecards! You can't tell the players without the scorecards!*—old chant of scorecard vendors at baseball games

*The foot bone connected to the ankle bone, The ankle bone connected to the shin bone...*—from the children's song, "Dem Bones"

The reason our unfortunate analyst in the preceding section found that his code ran more slowly on the parallel machine was almost certainly due to a lack of understanding of the underlying hardware and systems software. While one certainly need not understand the hardware on an electronics level, a basic knowledge of "what is connected to what" is essential.

In this section, we'll present overviews of the major hardware issues, and of the two parallel hardware technologies the reader is mostly likely to encounter, *multicore* and *clusters*[1]. We'll cover just enough details to illustrate the performance issues discussed later in this chapter, and return for further details in later chapters.

[1]What about clouds? A cloud consists of multicore machines and clusters too, but operating behind the scenese

### 2.2.1 Memory Basics

Slowness of memory access is one of the most common issues arising in high-performance computing. Thus a basic understanding of memory is vital.

Consider an ordinary assignment statement, copying one variable (a single integer, say) to another:

```
y = x
```

Typically, both **x** and **y** will be stored somewhere in memory, i.e. RAM (Random Access Memory). Memory is broken down into *bytes*, designed to hold one character, and *words*, usually designed to contain one number. A byte consists of eight bits, i.e. eight 0s and 1s. On typical computers today, the word size is 64 bits, or eight bytes.

Each word has an ID number, called an *address*. (Individual bytes have addresses too, but this will not concern us here.) So the compiler (in the case of C/C++/FORTRAN) or the interpreter in the case of a language like R), will assign specific addresses in memory at which **x** and **y** are to be stored.

A vector will typically be stored in a set of consecutive words. This will be the case for matrices too, but there is a question as to whether this storage will be row-by-row or column-by-column. C/C++ uses *row-major order*: First all of the first row (called row 0) is stored, then all of the second row, and so on. R and FORTRAN use *column-major order*, storing all of the first column (named column 1) etc. So, for instance, if **z** is a 5x8 matrix in R, then **z[2,3]** will be in the $12^{th}$ word (5+5+2) in the portion of memory occupied by **z**. These considerations will affect performance, as we will see later.

Memory access time, even though measured in tens of nanoseconds—billionths of a second—is slow relative to CPU speeds. This is due not only to electronic delays within the memory chips themselves, but also due to the fact that the pathway to memory is often a bottleneck. More on this below.

### 2.2.2 Caches

A device commonly used to deal with slow memory access is a *cache*. This is a small but fast chunk of memory that is located on or near the processor chip. For this purpose, memory is divided into *blocks*, say of 512 bytes each. Memory address 1200, for instance, would be in block 2, since 1200/512 is

equal to 2 plus a fraction. (The first block is called Block 0.) At any give time, the cache contains local copies of some blocks of memory, with the specific choice of blocks being dynamic—at some times the cache will contain copies of some memory blocks, while a bit later it may contain copies of some other blocks.[2]

If we are lucky, in most cases, the memory word that the processor wishes to access (i.e. the variable the programmer's code wishes to access) already has a copy in its cache—a *cache hit*. If this is a read access (of **x** in our little example above), then it's great—we avoid the slow memory access. If it is a write access (to **y** above), that's nice too, as it saves us the long trip to memory, but it does produce a discrepancy between the given word in memory and its copy in the cache. If we have just a single processor, i.e. a single core, as in a classic machine, we can tolerate this discrepancy, since no one else will be accessing that word in memory. Eventually the memory word must be updated, which will occur when this block is "evicted" (see blow), but again, this write hit means we avoid having to go to memory for now.

If the desired memory word is not currently in the cache, this is termed a *cache miss*. A cache read miss is fairly expensive. When it occurs, the entire block containing the requested word must be brought into the cache. In other words, we must access multiple words of memory, not just one. Moreover, usually a block currently in the cache must be *evicted* to make room for the new one being brought in. If the old block had been written to at all, we must now write that entire block back to memory, to update it.[3]

So, though we save memory access time when we have a cache hit, we incur a substantial penalty at a miss. Fortunately, the hardware usually does a good job, evicting the blocks that it thinks are least likely to be needed again in the near future. Cache hit rates are typically well above 90%. Note carefully, though, that this can be affected by the way we code. This will be discussed in future chapters.

### 2.2.3 Virtual Memory

Though it won't arise much in our context, we should at least briefly discuss *virtual memory*. Consider our example above, in which our program

[2]What follows below is a description of a common cache design. There are many variations, not discussed here.

[3]There is a *dirty bit* that records whether we've written to the block, but not which particular words were affected. Thus the entire block must be written.

contained variables **x** and **y**. Say these are assigned to addresses 200 and 8888, respectively. Fine, but what if another program is also running on the machine? The compiler/interpreter may have assigned one of its variables, say **g**, to address 200. How do we resolve this?

The standard solution is to make the address 200 (and all others) only "virtual." It may be, for instance, that **x** from the first program is actually stored in physical address 7260. The program will still say **x** is at word 200, but the hardware will translate 200 to 7260 as the program executes. If **g** in the second program is actually in word 6548, the hardware will replace 200 by 6548 every time the program requests access to word 200. The hardware has a table to do these lookups, one table for each program currently running on the machine, with the table being maintained by the operating system.

Virtual memory systems break memory into *pages*, say of 4096 bytes each, analogous to cache blocks. Usually, only some of your program's pages are *resident* in memory at any given time, with the remainder of the pages out on disk. If your program needs some memory word not currently resident—a *page fault*, analogous to a cache miss—the hardware senses this, and transfers control to the operating system. The OS must bring in the requested page from disk, an extremely expensive operation, due to the fact that a disk drive is mechanical rather than electronic like RAM. Thus page faults can really slow down program speed, and again as with the cache case, you may be able to reduce page faults through careful design of your code.

### 2.2.4 Monitoring Cache Misses and Page Faults

Both cache misses and page faults are enemies of good performance, so it would be nice to monitor them.

This actually can be done in the case of page faults. As noted, a page fault triggers a jump to the OS, which can thus record it. In Unix-family systems, the **time** command gives not only run time but also a count of page faults.

By contrast, cache misses are handled purely in hardware, thus not recordable by the OS. But one might try to gauge the cache behavior of a program by using the number of page faults as a proxy.

### 2.2.5 Locality of Reference

Clearly, the effectiveness of caches and virtual memory depend on repeatedly using items in the same blocks (*spatial locality*) within short time periods (*temporal locality*). As mentioned earlier, this in turn can be affected to some degree by the way the programmer codes things.

Say we wish to find the sum of all elements in a matrix. Should our code traverse the matrix row-by-row or column-by-column? In R, for instance, which as mentioned stores matrices in column-major order, we should go column-by-column, to get better locality.

A detailed case study will be presented in Section 5.17.

### 2.2.6 Latency and Bandwidth

The speed of a communications channel—whether between processor cores and memory in shared-memory platforms, or between network nodes in a cluster of machines–is measured in terms of *latency*—end-to-end travel time for a single bit—and *bandwidth*—the number of bits per second that we can pump onto the channel.

To make the notions a little more concrete, consider the San Francisco Bay Bridge, a long, mutlilane structure for which westbound drivers pay a toll. The notion of latency would desribe the time it takes for a car to drive from one end of the bridge to the other. (For simplificity, assume they all go the same speed.) By contrast, the bandwidth would be the number of cars exiting from the toll booths per unit time. We can reduce the latency by raising the speed limit on the bridge, while we could increase the bandwidth by adding more lanes and more toll booths.

The time in seconds to send an n-byte message, with a latency of l seconds and a bandwidth of b bytes/second, is clearly

$$l + n/b \tag{2.1}$$

Of course, this assumes that there are no other messages contending for the communication channel.

One way to ameliorate the slowdown from long latency delays is *latency hiding*. The basic idea is to try to do other useful work while a communication having long latency is pending. This approach is used, for instance, in the use of nonblocking I/O in message-passing systems (Section 4.5.1) to

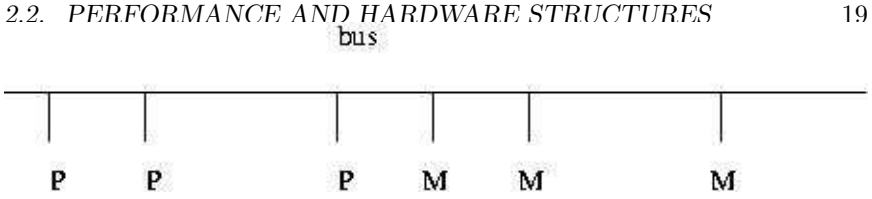


Figure 2.1: Symmetric Multiprocsssor System

deal with network latency, and in GPUs (Chapter 6) to deal with memory latency.

### 2.2.7 Two Representative Hardware Platforms: Multicore Machines and Clusters

Multicore machines have become standard on the desktop (even in the cell phone!), and many data scientists have access to computer clusters. What are the performance issues on these platforms? The next two sections provide an overview.

#### 2.2.7.1 Multicore

A *symmetric multiprocessor system* looks something like Figure 2.1 in terms of components and, most importantly, their interconnection. What do we see?

- There are *processors*, depicted by the Ps, in which your program is physically executed.
- There are *memory banks*, the Ms, in which your program and data reside during execution.
- The processors and memory banks are connected to a *bus*, a set of parallel wires used for communication between these computer components.

Your input/output hardware—disk drives, keyboards and so on—are also connected to the bus, and there may actually be more than one bus, but our focus will be mainly on the processors and memory.

A *threaded* program will have several instantiations of itself, called *threads*, that are working in concert to achieve parallelism. They run independently,

except that they share the data of the program in common. If your program is threaded, it will be running on several of the processors at once, each thread on a different core. A key point, as we will see, is that the shared memory becomes the vehicle for communication between the various processes.

Your program consists of a number of machine language instructions. (If you write in an interpreted language such as R, the interpreter itself consists of such instructions.) As the processors execute your program, they will fetch the instructions from memory.

As noted earlier, your data—the variables in your program—is stored in memory. The machine instructions fetch the data from memory as needed, so that it can be processed, e.g. summed, in the processors.

Until recently, ordinary PCs sold at your local electronics store followed the model in Figure 2.1 but with only one P. Multprocessor systems enabled parallel computation, bu cost hundreds of thousands of do;lars. But then it became standard for systems to have a *multicore* form. This means that there are multiple Ps, but with the important distinction that they are all one a single chip (each P is one core), making for inexpensive systems.[4] Whether on a single chip or not, having multiple Ps sets up parallel computation, and is known as the *shared memory* paradigm, for obvious reasons.

By the way, why are there multiple Ms in Figur 2.1? To improve memory performance, the system is set up so that memory is partitioned into several banks (typically there are the same number of Ms as Ps). This enables us to not only do *computation* on a parallel basis—several Ps working on different pieces of a problem in parallel—but also to do *memory access* in parallel—several memory accesses being active in parallel, in different banks. This amortizes the memory access penalty. Of course, if more than one P happens to need to access the same M at about the same time, we lose this parallelism.

As you can see, a potential bottleneck is the bus. When more than one P needs to access memory at a time, even if to different banks, attempting to place memory access requests on the bus, all but one of them will need to wait. This *bus contention* can cause significant slowdown. Much more elaborate systems, featuring multiple communications channels to memory rather than just a bus, have also been developed and serve to aemliorate the bottleneck issue. Most readers of this book, however, are more likely

[4]Terminology is not standardized, unfortunately. It is common to refer to that chip as "the" processor, even though there actually are multiple processors inside.

to use a multicore system on a single memory bus.

You can see now why efficient memory access is so crucial factor in achieving high performance. There is one more tool to handle this that is vital to discuss here: Use of caches. Note the plural; in Figure 2.1, there is usually a C in between each P and the bus.

As with uniprocessor systems, a caching can bring a big win in performance. In fact, the potential is even greater with a multiprocessor system, since caching will now bring the additional benefit of reducing bus contention. Unfortunately, it also produces a new problem, *cache coherency*, as follows.[5]

Consider what happens upon a write hit. The problem is that other caches may have a copy of this word, so they are now invalid for that block. The hardware must now inform them that they are invalid, it does via the bus, thus incurring an expensive bus operation. Moreover, the next time this word (or for that matter, any word in this block) is requested at one of the other caches, there will be a read miss, again an expensive event.

Once again, proper coding on the programmer's part can sometimes ameliorate the cache coherency problem.

A final point on multicore structure: Even on a uniprocessor machine, one generally has multiple programs running concurrently. You might have your browser busy downloading a file, say, while at the same time you are using a photo processing application. With just a single processor, these programs will actually take turns running; each one will run for a short time, say 50 milliseconds, then hand off the processor to the next program, in a cyclic manner. (You as the user probably won't be aware of this directly, but you may notice the system as a whole slowing down.) Note by the way that if a program is doing a lot of input/output (e.g. file access), it is effectively idle during I/O times; as soon as it starts an I/O operation, it will relinquish the processor.

By contrast, on a multicore machine, you can have multiple programs running physically simultaneously (though of course they will still take turns if there are more of them than there are cores).

But say you have threaded program, for example with four threads and a machine with four cores. Then the four threads will run physically simultaneously (if there are no other programs competing with them). That of course is the entire point, to achieve parallelism.

[5] As noted earlier, there are variations of the structure described here, but this one is typical.

#### 2.2.7.2 Clusters

These are much simpler to describe, though with equally thorny performance obstacles.

The term *cluster* simply refers to a set of independent *processing elements* (PEs) or *nodes* that are connected by a local area network, such as the common Ethernet or the high-performance Infiniband. Each PE consists of a CPU and some RAM. The PE could be a full desktop computer, including keyboard, disk drive and monitor, but if it is used primarily for parallel computation, then just one monitor, keyboard and so on suffice for the entire system. A cluster may also have a special operating system, to coordinate assigning of user programs to PEs.

We will may have one computational process per PE (unless of course each PE is a multicore system, as is common). Communication between the processes occurs via the network. The latter aspect, of course, is where the major problems occur, so let's discuss how networks work.

A single Ethernet, say within a building, is called a *network*. The *Internet* is simply the interconnection of many networks–millions of them.

Say you direct the browser on your computer to go to the Cable Network News (CNN) home page, and you are located in San Francisco. Since CNN is headquartered in Atlanta, a *packet* of information will go from San Francisco to Atlanta. (Actually, it may not go that far, since Internet service providers (ISPs) often cache Web pages, but let's suppose that doesn't occur.) Actually, the packet's journey will be rather complicated:

- Your browser program will write your Web request to a *socket*. The latter is not a physical object, but rather a software interface from your program to the network.

- The socket software will form a packet from your request, which will then go through several layers of the *network protocol stack* in your OS. Along the way, the packet will grow, as more information is being added.

- Eventually the packet will reach your computer's network interface hardware, from which it goes onto the network.

- A *gateway* on the network will notice that the ultimate destination is external to this network, so the packet will be transferred to another network that the gateway is also attached to.

- Your packet will wend its way across the country, being sent from one network to the next.[6]

- When your packet reaches a CNN computer, it will now work its way *up* the levels of the OS, finally reaching the Web server program.

Clearly there are numerous delays, including the less-obvious ones incurred in traversing the layers of the OS. Such traversal involves copying the packet from layer to layer, and in cases of interest in this book, such copying can involve huge matrices and thus take a lot of time.

Though parallel computation is typically done within a network rather than across networks as above, many of those delays are still there. So, network speeds are much, much slower than processor speeds, both in terms of latency and bandwidth.

The latency in even a fast network such as Infiniband is on the order of microseconds, i.e. millionths of a second, which is eons compared the nanosecond level of execution time for a machine instruction in a processor. (Beware of a network that is said to be fast but turns out only to have high bandwidth, not also low latency.)

These issues will be discussed further in Chapter 4.

## 2.3 How Many Processes/Threads?

As mentioned earlier, it is customary in the R world to refer to each worker in a **snow** program as a process. A question that then arises is, how many processes should we run?

Say for instance we have a cluster of 16 nodes. Should we set up 16 workers for our **snow** program? The same issues arise with threaded programs, say with **Rdsm** or OpenMP (Chapter 5). On a quadcore machine, should we run 4 threads?

The answer is *not* automatically Yes to these questions. With a fine-grained program, using too many processes/threads may actually degrade performance, as the overhead may overwhelm the presumed advantage of throwing more hardware at the problem. So, one might actually use fewer cluster nodes or fewer cores than one has available.

[6] Run the **traceroute** command on your machine to see the exact path, though this can change over time.

On the other hand, one might try to *oversubscribe* the resources. As discussed earlier, a cache miss causes a considerably delay, and a page fault even more. This is time during which one of the nodes/cores will not be doing any computation, exacting an opportunity cost from performance. It may pay, then, to have "extra" threads for the program available to run.

## 2.4 Example: Mutual Outlink Problem

To make this concrete, let's measure times for the mutual outlinks problem, with larger and larger numbers of processes.

Here I ran on a shared memory machine consisting of four processor chips, each of which has eight cores. This gives us a 32-core system, and I ran the mutual outlinks problem with values of **nc**, the number of cores, equal to 2, 4, 6, 8, 10, 12, 16, 24, 28 and 32. The problem size was 1000 rows by 1000 columns. The times are plotted in Figure 2.2.

Here we see a classical U-shaped pattern: As we throw more and more processes on the problem, it helps in the early stages, but performance actually degrades when after a certain point. The latter phenomenon is probably due to the communications overhead we discussed earlier, in this case cache issues, memory contention and so on.

By the way, for each of our **nc** workers, we had one invocation of R running on the machine. There was also an additional invocation, for the manager. However, this is not a performance issue in this case, as the manager spends most of its time idle, waiting for the workers.

## 2.5 "Big O" Notation

With all this talk of physical obstacles to overcome, such as memory access time, it's important also to raise the question as to whether the application itself is very parallelizable in the first place. One measure of that is "big O" notation.

In our mutual outlinks example with an nxn adjacency matrix, we need to do on average n/2 sum operations per row, with n rows, thus $n \cdot n/2$ operations in all. In parallel processing circles, the key question asked about hardware, software, algorithms and so on is, "Does it scale?", meaning, Does the run time grow manageably as the problem size grows?

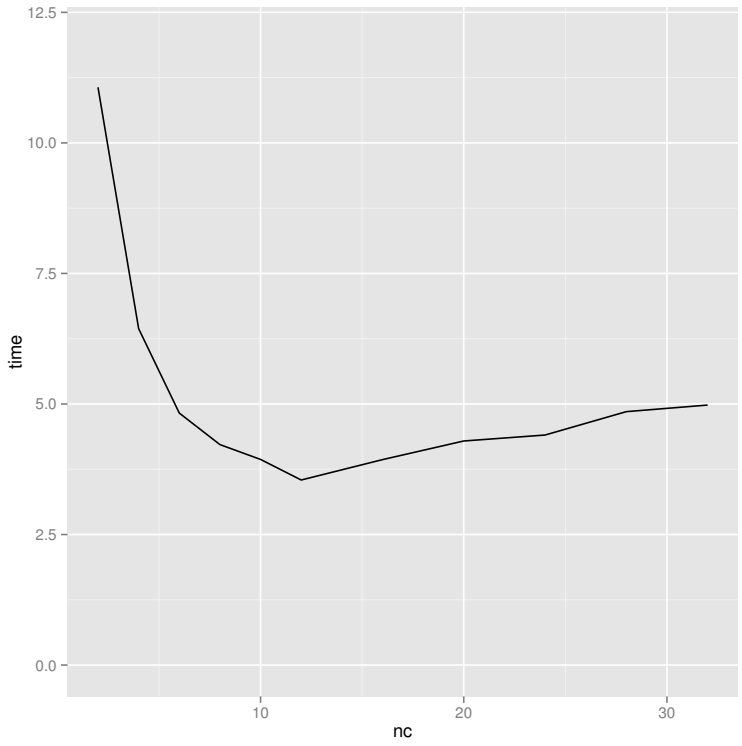


Figure 2.2: Run Time Versus Number of Cores

We see above that the run time of the mutual outlinks problem grows proportionally to the *square* of the problem size. (Dividing by 2 doesn't affect this growth rate.) We write this as $O(n^2)$, known colloquially as "big O" notation. When applied to analysis of run time, we say that it measures the *time complexity*.

Ironically, application that *are* manageable often are poor candidates for parallel processing, due to overhead playing a greater role in such problems. An application with $O(n)$ time complexity, for instance, may present a challenge. We will return to this notion at various points in this book.

## 2.6 Data Serialization

Some parallel R packages, e.g. **snow**, that send data through a network *serialize* the data, meaning to convert it to ASCII form. The data must then be unserialized on the receiving end. This creates a delay, which may or may not be serious but must be taken into consideration.

## 2.7 "Embarrassingly Parallel" Applications

The term *embarrassingly parallel* is heard often in talk about parallel programming.

### 2.7.1 What People Mean by "Embarrassingly Parallel"

Consider a matrix multiplication application, for instance, in which we compute AX for a matrix A and a vector X. One way to parallelize this problem would be to have each processor handle a group of rows of A, multiplying each by X in parallel with the other processors, which are handling other groups of rows. We call the problem embarrassingly parallel, with the word "embarrassing" meaning that the problem is too easy, i.e. there is no intellectual challenge involved. It is pretty obvious that the computation Y = AX can be parallelized very easily by splitting the rows of A into groups.

By contrast, most parallel sorting algorithms require a great deal of interaction. For instance, consider Mergesort. It breaks the vector to be sorted into two (or more) independent parts, say the left half and right half, which

are then sorted in parallel by two processes. So far, this is embarrassingly parallel, at least after the vector is broken in half. But then the two sorted halves must be merged to produce the sorted version of the original vector, and that process is *not* embarrassingly parallel; it can be parallelized, but in a more complex, less obvious manner.

Of course, it's no shame to have an embarrassingly parallel problem! On the contrary, except for showoff academics, having an embarrassingly parallel application is a cause for celebration, as it is easy to program.

In recent years, the term *embarrassingly parallel* has drifted to a somewhat different meaning. Algorithms that are embarrassingly parallel in the above sense of simplicity tend to have very low communication between processes, key to good performance. That latter trait is the center of attention nowadays, so the term *embarrassingly parallel* generally refers to an algorithm with low communication needs.