13
Computer Hardware Basics: Memory and CPU

In this chapter we discuss hardware aspects that are important for high performance computing (HPC). While it may not seem that what we do in this book can be called HPC, history keeps showing that what is HPC today will be on everyone's desktop or laptop in less than a decade. Indeed, the last 30 years have seen a computer progression: scalar → superscalar → vector → parallel, and these are topics that we will talk about here. More recent developments, such as programming for multicore computers, cell computers, and field programmable gate accelerators, will be discussed in future books.

In HPC, you generally modify or "tune" your program to take advantage of a computer's architecture. Often the real problem is to determine which parts of your program get used the most and to decide whether they would run significantly faster if you modified them to take advantage of a computer's architecture. In this chapter we mainly discuss the theory of a high-performance computer's memory and central processor design. In Chap. 13, we concentrate on how you determine the most numerically intensive parts of your program, and how specific hardware features affect them.

Your **problem** is to make a numerically intensive program run faster, but not by porting it to a faster computer. We assume that you are already running your programs on a scientific computer, and so need to know how to make better use of it. In this chapter we discuss the hardware of high-performance computers, while in Chap. 14 we apply that knowledge to your problem.

13.1
High-Performance Computers (CS)

By definition, supercomputers are the fastest and most powerful computers available, and are the superstars of the high-performance class of computers. At this instant, "supercomputers" almost always refer to parallel machines. Unix workstations and modern personal computers (PCs), which are small enough in size and cost to be used by a small group or an individual, yet powerful enough for large-scale scientific and engineering applications, can

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

ISBN: 978-3-527-40626-5

also be high-performance computers. We define high-performance computers as machines with good balance among the following major elements:

- multistaged (pipelined) functional units
- multiple central processing units (parallel machines)
- fast, central registers
- very large and fast memories
- very fast communication among functional units
- vector or array processors
- software that integrates the above effectively.

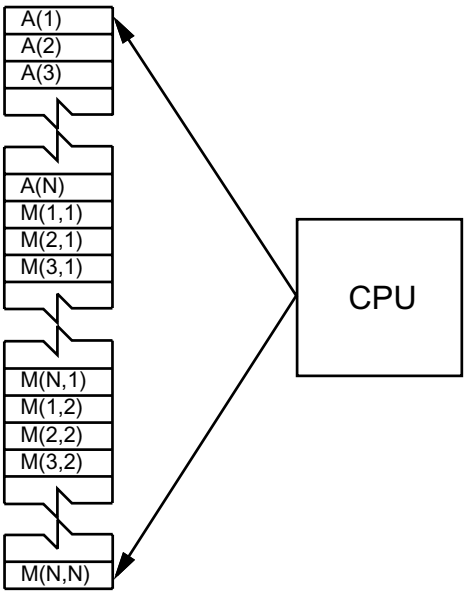

Fig. 13.1 The logical arrangement of CPU and memory showing a Fortran array, $A(N)$, and matrix $M(N,N)$ loaded into memory.

13.1.1 Memory Hierarchy

An idealized model of computer architecture is a CPU sequentially executing a stream of instructions and reading from a continuous block of memory. To illustrate, in Fig. 13.1 we see a vector `A[]` and an array `M[][]` loaded in memory and about to be processed. The real world is more complicated than

this. First, matrices are not stored in blocks, but rather in linear order. For instance, in Fortran it would be in *column-major* order:

```
M(1,1) M(2,1) M(3,1) M(1,2) M(2,2) M(3,2) M(1,3) M(2,3)
M(3,3),
```

while in Java and C it would be in *row-major* order:

```
M(0,0) M(0,1) M(0,2) M(1,0) M(1,1) M(1,2) M(2,0) M(2,1)
M(2,2).
```

Second, the values for the matrix elements may not even be in the same physical memory (Fig. 13.2). Some may be in RAM, some on the disk, some in cache, and some in the CPU.

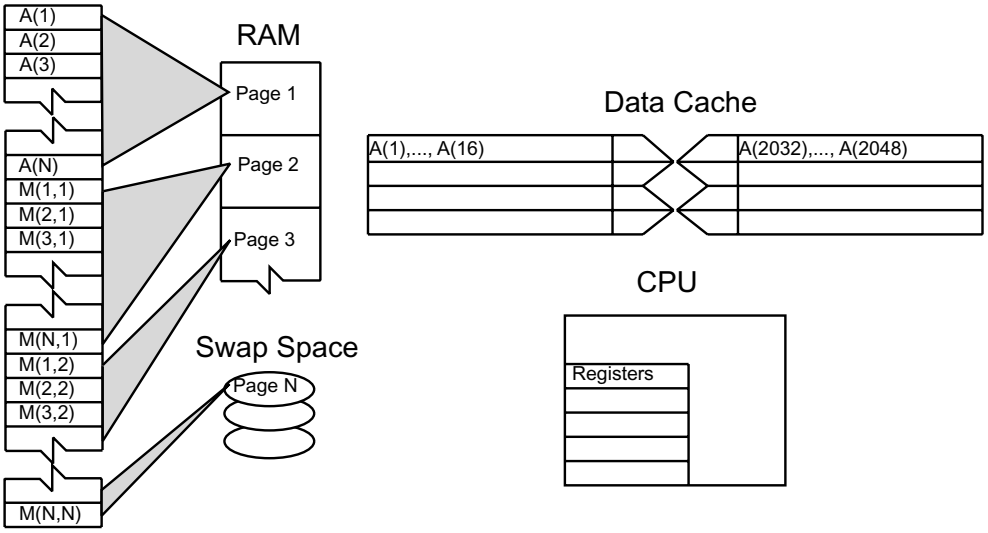

Fig. 13.2 The elements of a computer's memory architecture in the process of handling matrix storage.

To give some of these words more meaning, in Fig. 13.3 we show a simple model of the complex memory architecture of a high-performance computer. This hierarchical arrangement arises from an effort to balance speed and cost, with fast, expensive memory, supplemented by slow, less expensive memory. The memory architecture may include the following elements:

CPU: Central processing unit, the fastest part of the computer. The CPU consists of a number of very high-speed memory units called *regis-*

ters, containing the *instructions* sent to the hardware to do things like fetch, store, and operate on data. There are usually separate registers for instructions, addresses, and *operands* (current data). In many cases the CPU also contains some specialized parts for accelerating the processing of floating point numbers, something that used to be done with a separate piece of hardware.

Cache: A small, very fast bit of memory also called the *high-speed buffer* holds instructions, addresses, and data in their passage between the very fast CPU registers and the slower main RAM memory. This is seen in the next level down the pyramid in Fig. 13.3. The main memory is also called *dynamic RAM* (DRAM), while the cache is *static RAM* (SRAM). If the cache is used properly, it eliminates the need for the CPU to wait for data to be fetched from memory.

Cache and data lines: The data transferred to and from the cache or CPU are grouped into *cache lines* or *data lines*. The time it takes to bring data from memory into cache is called *latency*.

RAM: Random access memory or central memory is in the middle memory hierarchy in Fig. 13.3. RAM can be accessed directly (i.e., in random order), and it can be accessed quickly (i.e., without mechanical devices). It is where your program resides while it is being processed.

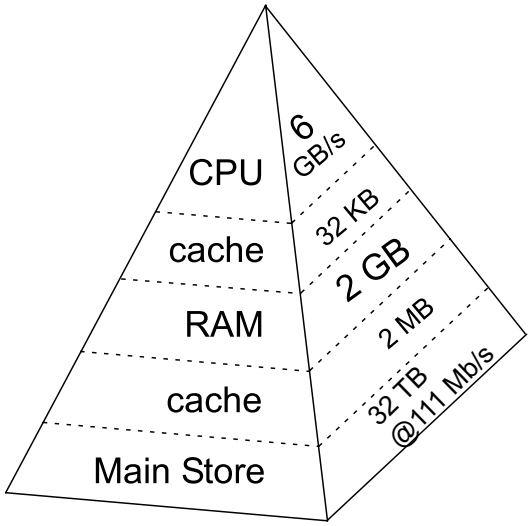

Fig. 13.3 Typical memory hierarchy for a single-processor high-performance computer (B = bytes, K, M, G, T = kilo, mega, giga, terra).

Pages: Central memory is organized into *pages*, which are blocks of memory of fixed length. The operating system labels and organizes its memory pages much like we do the pages of books; they are numbered and kept track of with a *table of contents*. Typical page sizes are from 4–16 kB.

Hard disk: Finally, at the bottom of the memory pyramid is the permanent storage on magnetic disks or optical devices. Although disks are very slow compared to RAM, they can store vast amounts of data and sometimes compensate for their slower speeds by using a cache of their own, the *paging storage controller*.

Virtual memory: True to its name, this is a part of memory you will not find in our figures because it is *virtual*. It acts like RAM, but resides on the disk.

When we speak of "fast" and "slow" memory we are using a time scale set by the clock in the CPU. To be specific, if your computer has a clock speed or cycle time of 1 ns, this means that it could perform a billion operations per second, if it could get its hands on the needed data quickly enough (typically, more than 10 cycles are needed to execute a single instruction). While it usually takes 1 cycle to transfer data from the cache into the CPU, the other memory is much slower, and so you can speed your program up by not having the CPU wait for transfers among different levels of memory. Compilers try to do this for you, but their success is affected by your programming style.

As shown in Fig. 13.2 for our example, virtual memory permits your program to use more pages of memory than will physically fit into RAM at one time. A combination of operating system and hardware *maps* this virtual memory into pages with typical lengths of 4–16 kB. Pages not currently in use are stored in the slower memory on the hard disk and brought into fast memory only when needed. The separate memory location for this switching is known as *swap space* (Fig. 13.2). Observe that when the application accesses the memory location for `M[i][j]`, the number of the page of memory holding this address is determined by the computer, and the location of `M[i][j]` within this page is also determined. A *page fault* occurs if the needed page resides on the disk rather than in RAM. In this case the entire page must be read into memory while the least-recently used page in RAM is swapped onto the disk.

Thanks to virtual memory, it is possible to run programs on small computers that otherwise would require larger machines (or extensive reprogramming). The price you pay for virtual memory is an order-of-magnitude slowdown of your program's speed when virtual memory is actually invoked. But this may be cheap compared to the time you would have to spend to rewrite

your program so it fits into RAM, or the money you would have to spend to buy a computer with enough RAM for your problem.

Virtual memory also allows *multitasking*, the simultaneous loading into memory of more programs than will physically fit into RAM. Although the ensuing switching among applications uses computing cycles, by avoiding long waits while an application is loaded into memory, multitasking increases the total throughout and permits an improved computing environment for users. For example, it is multitasking that permits windows system to provide us with multiple windows. Even though each window application uses a fair amount of memory, only the single application currently receiving input must actually reside in memory; the rest are *paged out* to disk. This explains why you may notice a slight delay when switching to an idle window; the pages for the now-active program are being placed into RAM and simultaneously the least-used application still in memory is paged out.

13.2 The Central Processing Unit

How does the CPU get to be so fast? Often, it employs *prefetching* and *pipelining*; that is, it has the ability to prepare for the next instruction before the current one has finished. It is like an assembly line or a bucket brigade in which the person filling the buckets at one end of the line does not wait for each bucket to arrive at the other end before filling another bucket. In this same way a processor fetches, reads, and decodes an instruction while another instruction is executing. Consequently, even though it may take more than one cycle to perform some operations, it is possible for data to be entering and leaving the CPU on each cycle. To illustrate, consider how the operation $c = (a + b)/(d * f)$ is handled (see Tab. 13.1):

Tab. 13.1 Computation of $c = (a + b)/(d * f)$.

Arithmetic unit	Step 1	Step 2	Step 3	Step 4
A1	Fetch a	Fetch b	Add	—
A2	Fetch d	Fetch f	Multiply	—
A3	—	—	—	Divide

Here the pipelined arithmetic units A1 and A2 are simultaneously doing their jobs of fetching and operating on operands, yet arithmetic unit A3 must wait for the first two units to complete their tasks before it gets something to do (during which time the other two sit idle).

13.2.1 CPU Design: RISC

RISC is an acronym for Reduced Instruction Set Computer (also called superscalar). It is a design philosophy for the CPU's architecture developed for high-performance computers and now used broadly. It increases the arithmetic speed of the CPU by decreasing the number of instructions the CPU must follow.

To understand RISC we contrast it with *CISC*, Complex Instruction Set Computers. In the late 1970s, processor designers began to take advantage of Very Large Scale Integration *VLSI* which allowed the placement of hundreds of thousands of devices on a single CPU chip. Much of the space on these early chips was dedicated to *microcode* programs written by the chip designers and containing machine language instructions that set the operating characteristics of the computer. There were over 1000 instructions available, and many were similar to higher level programming languages such as *Pascal* and *Forth*. The price paid for the large number of complex instructions was slow speed, with a typical instruction taking more than 10 clock cycles. Furthermore, a 1975 study by Alexander and Wortman of the *XLP* compiler of the IBM System/360 showed that 10 low-level machine instructions accounted for 80% of the use, while some 30 low-level instructions accounted for 99% of the use.

The RISC philosophy is to have just a small number of instructions available at the chip level, but to have the regular programmer's high-level language, such as Fortran or C, translate them into efficient machine instructions for a particular computer's architecture. This simpler scheme is cheaper to design and produce, lets the processor run faster, and uses the space saved on the chip by cutting down on microcode to increase arithmetic power. Specifically, RISC increases the number of internal CPU registers, thus making it possible to obtain longer pipelines (cache) for the data flow, a significantly lower probability of memory conflict, and some instruction-level parallelism.

The theory behind this philosophy for RISC design is the simple equation describing the execution time of a program

$$CPU\ time = \#\ instructions \times cycles/instruction \times cycle\ time \tag{13.1}$$

Here $CPU\ time$ is the time required by a program; $\#\ instructions$ is the total number of machine-level instructions the program requires (sometimes called the *path length*); $cycles/instruction$ is the number of CPU clock cycles each instruction requires; and *cycle time* is the actual time it takes for 1 CPU cycle. After viewing (13.1) we can understand the CISC philosophy which tries to reduce *CPU time* by reducing # *instructions*, as well as the RISC philosophy which tries to reduce *CPU time* by reducing the *cycles per instruction* (preferably to one). For RISC to achieve an increase in performance requires a greater

decrease in cycle time and cycles/instruction than the increase in the number of instructions.

In summary, the elements of RISC are:

[**Single-cycle execution**] for most machine-level instructions.

[**Small instruction set**] of less than 100 instructions.

[**Register-based instructions**] operating on values in registers, with memory access confined to load and store to and from registers.

[**Many registers,**] usually more than 32.

[**Pipelining,**] that is, concurrent processing of several instructions.

[**High level compilers**] to improve performance.

13.2.2 CPU Design: Vector Processor

Often the most demanding part of a scientific computation involves matrix operations. On a classic (von Neumann) scalar computer, the addition of two vectors of physical length 99 to form a third ultimately requires 99 sequential additions (see Tab. 21.1):

Tab. 13.2 Computation of matrix $[C] = [A] + [B]$.

Step 1	Step 2	$\cdots$	Step 99
$c(1) = a(1) + b(1)$	$c(2) = a(2) + b(2)$	$\cdots$	$c(99) = a(99) + b(99)$

There is actually much behind-the-scene work here. For each element i there is the *fetch* of $a(i)$ from its location in memory, the *fetch* of $b(i)$ from its location in memory, the addition of the numeric values of these two elements in a CPU register, and the *storage* in memory of the sum into $c(i)$. This fetching uses up time and is wasteful in the sense that the computer is being told again and again to do the same thing.

When we speak of a computer doing "vector" processing, we mean that there are hardware components that perform mathematical operations on entire rows or columns of "matrices" as opposed to individual elements. (This hardware also can handle single-subscripted matrices, that is, mathematical vectors.) In *vector* processing of $[A] + [B] = [C]$, the successive fetching and additions of the elements of A and B get grouped together and overlaid, and

$Z \simeq 64$–256 elements (the *section size*) are processed with one command (see Tab. 13.3):

Tab. 13.3 Vector processing of matrix $[A] + [B] = [C]$.

Step 1	Step 2	Step 3	$\cdots$	Step Z
$c(1) = a(1) + b(1)$				
	$c(2) = a(2) + b(2)$			
		$c(3) = a(3) + b(3)$		
			$\cdots$	
				$c(Z) = a(Z) + b(Z)$

Depending on the array size, this vector processing may speed up the processing of vectors by a factor of about 10. If all Z elements were truly processed in the same step, then the speedup would be ~ 64–256.

Vector processing probably had its heyday during the time when computer manufactures produced large mainframe computers designed for the scientific and military community. These computers had proprietary hardware and software, and were often so expensive that only corporate or military laboratories could afford them. While the Unix and then PC revolutions have nearly eliminated these large vector machines, some do exist, as well as PCs that use vector processing in their video cards. Who is to say what the future may hold in store?