8

Performance Basics; Computer Architectures

8.1 Speed and limiting factors of computations

Basic floating-point operations, such as addition and multiplication, are carried out directly on the central processor unit (CPU). Elementary functions, on the other hand, are emulated on a higher level. Table 8-I provides an overview of the typical execution times for basic mathematical operations.

integer addition, subtraction, or multiplication	<1
integer division	4–10
float addition, subtraction, or multiplication	1
float division	2–6
sqrt	5–20
sin, cos, tan, log, exp	10–40

Table 8-I: *The relative speed of integer operations, floating-point operations, and several elementary functions. (This is based on C and Fortran programs compiled with various compilers and executed on various platforms.)*

(A simple interactive exercise with a program that adds a number a billion times to a variable can be used to approximately determine relative and absolute execution times. Such an exercise reveals the ratios above. It also reveals that Python is far slower than Fortran.)

```
> gfortran speed.f90        % 10^9 additions
> time a.out
3.846u 0.007s
```

```
> gfortran -O speed.f90      % 10^9 additions with optimization
> time a.out
1.306u 0.003s
> time python speed.py     % 10^8 additions
19.949u 16.576s
```

A unit of 1 in table 8-I corresponded to about 1 nanosecond on a personal computer or workstation (with a clock cycle of about 3 GHz). Contemplate however how many multiplications can be done in one second: 10^9. One second is enough time to solve a linear system in about 1000 variables or to take the Fourier Transform of a million points. Arithmetic is fast!

Even the relative speeds in the above table are much unlike doing such calculations with paper-and-pencil. On a computer additions are no faster than multiplications, unlike when done by hand. On modern processors multiplication takes hardly longer than addition and subtraction. This was not always so and it was appropriate to worry more about the number of multiplications than about the number of additions. If a transformation would replace a multiplication by several additions, it was favorable for speed, but this is no longer the case.

Over the decades the bottlenecks for programs have changed with technology. Long ago it was memory size. Memory was expensive compared to floating-point operations, and algorithms tried to save every byte of memory possible. Later, memory became comparatively cheap, and the bottleneck moved to floating-point operations. For example, storing repeatedly used results in a temporary variable rather than recalculating them each time increased speed. This paradigm of programming is the basis of classical numerical analysis, with algorithms designed to minimize the number of floating-point operations. Today the most severe bottleneck often is moving data from memory to the processor. In future, perhaps, the bottleneck will be the parallelizability of an algorithm.

Moore's law is a statement about the number of transistors per area that can be fit on a chip, and combined with other improvements, it has translated into an exponential and dramatic increase in floating point operations per second (FLOPS) over many decades. Until about the mid-2000's, processor performance had doubled about every 18–24 months.

Since then, the rate of increase for FLOPS per microprocessor has slowed, causing a sea change toward multi-core processors. This ushers in a new era for parallel computing.

There are basically four limiting factors to computations:

- processor speed
- memory size
- data transfer between memory and processor
- input/output

8.2 Memory and data transfer

The memory occupied by a number can be machine dependent, but standardization has led to significant uniformity. Each data type takes up a fixed number of bytes,

- four bytes for an integer,
- four bytes for a single precision number, and
- eight bytes for a double precision number.

Hence the required memory can be precisely calculated. For example, an array of 1024×1024 double-precision numbers takes up exactly eight megabytes ($2^{10} \approx 10^3$).

CPU		1 ns
Memory (DRAM)		50–70 ns
Harddrive	Solid-state	0.1 ms = 10^5ns
	Magnetic disk	5–10 ms = 10^7ns

Table 8-II: *Execution and access times for basic computer components. Most of these numbers are cited from Patterson & Hennessy (2013).*

Table 8-II shows how critical the issue of data transfer is, both between processor and memory and between memory and hard disk. When a processor carries out instructions it first needs to fetch necessary data from the memory. This is a slow process, compared to the speed with which the processor is able to compute. To speed up the transport of data a "cache" (pronounced "cash") is used, which is a small unit of fast

memory nowadays located on the main chip. A hierarchy of several levels of caches is possible, and in fact customary. Frequently used data are stored in the cache to be quickly accessible for the processor. Data are moved from main memory to the cache not byte by byte but in larger units of "cache lines," assuming that nearby memory entries are likely to be needed by the processor soon (assumption of "spatial locality"). Similarly, "temporal locality" assumes if a data location is referenced then it will tend to be referenced again soon. If the processor requires data not yet in the cache, one speaks of "cache misses," which lead to a time delay.

Table 8-III provides an overview of the memory hierarchy and the relative speed of its components. The large storage media are slow to access. The small memory units are fast.

Registers	1
Level 1 Cache	2–4
Main memory	60
Magnetic disk	10^7

Table 8-III: *Memory hierarchy and relative access times (clock cycles). See table 8-II for comparison with CPU execution speed.*

Programs can access more (virtual) memory than physically exists on the computer. If the data exceed the available memory, the harddrive is used for temporary storage (swap space). Since reading and writing from and to a harddrive is comparatively slow, this slows down the calculation dramatically.

Reading or writing a few bytes to or from a magnetic disk takes as long as millions of floating-point operations. The majority of this time is for the head, that reads and writes the data, to find and move to the location on the disk where the data are stored. Consequently data should be read and written in big junks rather than in small pieces. In fact the computer will try to do so automatically. While a program is executing, data written to a file may not appear immediately. The data are not flushed to the disk until they exceed a certain size or until the file is closed.

Non-volatile flash (solid-state) memory often replaces a magnetic disk as a hard drive. It is faster (and more expensive per byte) than magnetic drives, but wears out with time.

Input and output are relatively slow on any medium (magnetic hard-disk, screen, network, etc.). Writing on the screen is a particularly slow process; excesses thereof can easily delay the program. A common beginner's mistake is to display vast amounts of output on the screen, so that data scroll down the screen at high speed; this slows down the calculation.

Input/output can be limiting due to the data transfer rate, but also due to size. Large data will be discussed in chapter 14.

8.3 A programmer's view of computer hardware

In this section we look at how a program is processed by the computer and follow it from the source code down to the level of individual bits executed on the hardware.

The lines of written program code are ultimately translated into hardware dependent machine code. For instance, the following simple line of C code adds two variables: `a=i+j`. Suppose `i` and `j` have earlier been assigned values and are stored in memory. At a lower level we can look at the program in terms of its "assembly language," which is a symbolic representation of the binary sequences the program is translated into:

```
lw $8, i
lw $9, j
add $10, $8, $9
sw $10, a
```

The values are pulled from main memory to a small memory unit on the processor, called "register," and then the addition takes place. In this example, the first line loads variable `i` into register 8. The second line loads variable `j` into register 9. The next line adds the contents of registers 8 and 9 and stores the result in register 10. The last line copies the content of register 10 to memory. There are typically about 32 registers; they store only a few hundred bytes. Arithmetic operations,

in fact most instructions, operate not directly on entries in memory but on entries in the registers.

At the assembly language level there is no distinction between data of different types. Floating-point numbers, integers, characters, and so on are all represented as binary sequences. What number actually corresponds to the sequence is a matter of how it is interpreted by the instruction. There is a different addition operation for integers and floats, for example.

Instructions themselves, like `lw` and `add`, are also encoded as binary sequences. The meaning of these sequences is hardware-encoded on the processor (instruction set). When a program is started, it is first loaded into memory. Then the instructions are executed with clock cycles.

During consecutive clock cycles the processor needs to fetch the instruction, read the registers, perform the operation, and write to the register. Depending on the actual hardware these steps may be split up into even more substeps. The idea of "pipelining" is to execute every step on a different, dedicated element of the hardware. The next instruction is already fetched, while the previous instruction is at the stage of reading registers, and so on. Effectively, several instructions are processed simultaneously. Hence, even a single processor core tries to execute tasks in parallel, an example of instruction-level parallelism.

The program is stalling when the next instruction depends on the outcome of the previous one, as for a conditional statement. Although an `if` instruction itself is no slower than other elementary operations, it can stall the program in this way. In addition, an unexpected jump in the program can lead to cache misses. For the sake of speed, the programmer should keep the data flow predictable.

A computer's CPU is extremely complex. For instance, the processor uses its own intelligence to decide what data to store in a register. And for conditional operations, it may speculate which of the possible branches is most likely to occur next. Much of this complexity is hidden from the user, even at the assembly language level, and taken care of automatically.

Recommended Reading: Patterson & Hennessy, *Computer Organiza-*

```
do i=1,1000000000
   a=a+12.1
   a=a+7.8
end do
```

```
do i=1,1000000000
   a=a+12.1
   b=b+7.8
end do
```

Figure 8-1: *Although both of these Fortran loops involve two billion additions, the version to the right is twice as fast.*

tion and Design: The Hardware/Software Interface is a very accessible textbook for this rapidly changing field by two eminent scientists. Updated editions are being published regularly.