1

Introduction – The Nature of High-Performance Computation

The need for speed. Since the beginning of the era of the modern digital computer in the early 1940s, computing power has increased at an exponential rate (see Fig. 1). Such an exponential growth is predicted by the well-known "Moore's Law," first advanced in 1965 by Gordon Moore of Intel, asserting that the number of transistors per inch on integrated circuits will double every 18 months. Clearly there has been a great need for ever more computation. This need continues today unabated. The calculations performed by those original computers were in the fields of ballistics, nuclear fission, and cryptography. And, today these fields, in the form of computational fluid dynamics, advanced simulation for nuclear testing, and cryptography, are among computing's Grand Challenges.

In 1991, the U.S. Congress passed the High Performance Computing Act, which authorized The Federal High Performance Computing and Communications (HPCC) Program. A class of problems developed in conjunction with the HPCC Program was designated "Grand Challenge Problems" by Dr. Ken Wilson of Cornell University. These problems were characterized as "fundamental problems in science and engineering that have broad economic or scientific impact and whose solution can be advanced by applying high performance computing techniques and resources." Since then various scientific and engineering committees and governmental agencies have added problems to the original list. As a result, today there are many Grand Challenge problems in engineering, mathematics, and all the fundamental sciences. The ambitious goals of recent Grand Challenge efforts strive to

- build more energy-efficient cars and airplanes,
- design better drugs,
- forecast weather and predict global climate change,
- improve environmental modeling,

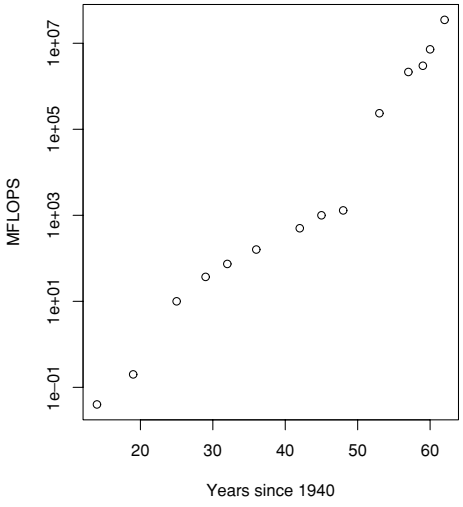

Fig. 1. Computational speed in MFLOPS vs. year.

- improve military systems,
- understand how galaxies are formed,
- understand the nature of new materials, and
- understand the structure of biological molecules.

The advent of high-speed computation has even given rise to computational subfields in some areas of science and engineering. Examples are computational biology, bioinfomatics, and robotics, just to name a few. Computational chemistry can boast that in 1998 the Noble Prize in chemistry was awarded to John Pope and shared with Walter Kohn for the development of computational methods in quantum chemistry.

And so it seems that the more computational power we have, the more use we make of it and the more we glimpse the possibilities of even greater computing power. The situation is like a Moore's Law for visionary computation.

1.1 Computing Hardware Has Undergone Vast Improvement

A major factor in the exponential improvement in computational power over the past several decades has been through advances in solid-state physics: faster switching circuits, better heat control, faster clock rates, faster memory. Along with advances in solid-state physics, there has also been an evolution in the architecture of the computer itself. Much of this revolution was spearheaded by Seymour Cray.

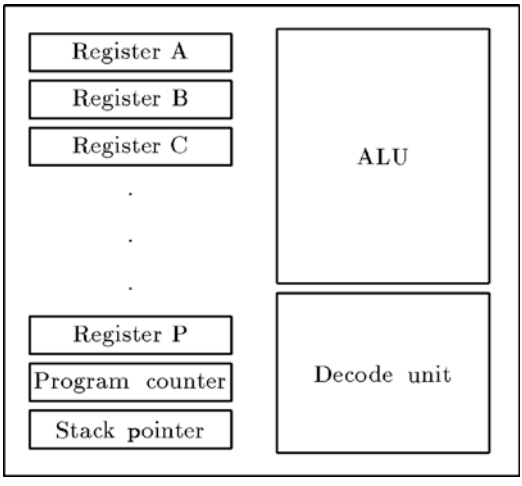

Fig. 2. Central processing unit.

Many ideas for parallel architectures have been tried, tested, and mostly discarded or rethought. However, something is learned with each new attempt, and the successes are incorporated into the next generation of designs. Ideas such as interleaved memory, cache memory, instruction look ahead, segmentation and multiple functional units, instruction piplining, data pipelining, multiprocessing, shared memory, distributed memory have found their way into the various catagories of parallel computers available today. Some of these can be incorporated into all computers, such as instruction look ahead. Others define the type of computer; thus, vector computers are data pipelined machines.

The von Neumann Computer

For our purposes here, a computer consists of a central processing unit or *CPU*, memory for information storage, a path or *bus* over which data flow and a synchronization mechanism in the form of a *clock*. The CPU itself consists of several internal registers – a kind of high-speed memory, a program counter (PC), a stack pointer (SP), a decode unit (DU), and an arithmetic and logic unit (ALU) (see Fig. 2). A program consists of one or more contiguous memory locations, that is, chunks of memory, containing a *code segment* including subroutines, a *data segment* for the variables and parameters of the problem, a *stack segment*, and possibly additional memory allocated to the program at run time (see Fig. 3).

The various hardware elements are synchronized by the clock whose frequency f characterizes the speed at which instructions are executed. The

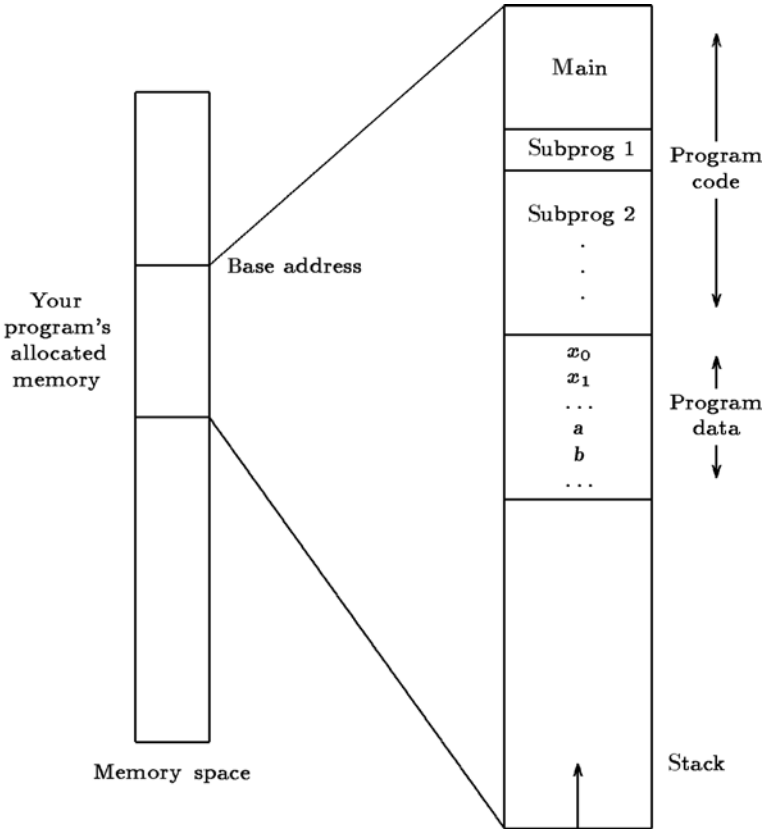

Fig. 3. Organization of main memory.

frequency is the number of cycles of the clock per second measured in megaHertz (mHz), 1 mHz $= 10^6$ Hz or gigaHertz, (gHz), 1 gHz $= 10^9$ Hz. The time t for one clock cycle is the reciprocal of the frequency

$$t = \frac{1}{f}.$$

Thus a 2-ns clock cycle corresponds to a frequency of 500 mHz since 1 ns $=$ 10^{-9} s and

$$f = \frac{1}{2 \times 10^{-9}} = 0.5 \times 10^9 = 500 \times 10^6.$$

If one instruction is completed per clock cycle, then the instruction rate, *IPS*, is the same as the frequency. The instruction rate is often given in millions of

instructions per second or *MIPS*; hence, *MIPS* equals megaHertz for such a computer.

The original computer architecture, named after John von Neumann, who was the first to envision "stored programming" whereby the computer could change its own course of action, reads instructions one at a time sequentially and acts upon data items in the same way. To gain some idea of how a von Neumann computer works, we examine a step-by-step walk-through of the computation $c = a + b$.

Operation of a von Neumann Computer: $c = a + b$ Walk-Through

On successive clock cycles:

Step 1. Get next instruction
Step 2. Decode: fetch a
Step 3. Fetch a to internal register
Step 4. Get next instruction
Step 5. Decode: fetch b
Step 6. Fetch b to internal register
Step 7. Get next instruction
Step 8. Decode: add a and b (result c to internal register)
Step 9. Do the addition in the ALU (see below)
Step 10. Get next instruction
Step 11. Decode: store c (in main memory)
Step 12. Move c from internal register to main memory

In this example two *floating point number*s are added. A floating point number is a number that is stored in the computer in mantissa and exponent form (see Section 4.1); integer numbers are stored directly, that is, with all mantissa and no exponent. Often in scientific computation the results materialize after a certain number of *floating point operation*s occur, that is, additions, subtractions, multiplications, or divisions. Hence computers can be rated according to how many floating point operations per second, or FLOPS, they can perform. Usually it is a very large number and hence measured in mega-FLOPS, written MFLOPS, or giga-FLOPS written GFLOPS, or tera-FLOPS (TFLOPS). Of course, 1 MFLOPS $= 10^6$ FLOPS, 1 GFLOPS $= 10^3$ MFLOPS $= 10^9$ FLOPS, and 1 TFLOPS $= 10^{12}$ FLOPS.

The addition done at step 9 in the above walk-through consists of several steps itself. For this illustration, assume 0.9817×10^3 is to be added to 0.4151×10^2.

Step 1. Unpack operands: 9817 | 3 4151 | 2
Step 2. Exponent compare: 3 vs. 2

Step 3. Mantissa align: 9817 | 3 0415 | 3
Step 4. Mantissa addition: 10232 | 3
Step 5. Normalization (carry) check: $\boxed{1}$0232 | 3
Step 6. Mantissa shift: 1023 | 3
Step 7. Exponent adjust: 1023 | 4
Step 8. Repack result: 0.1023×10^4

So if the clock speed is doubled, then each computer instruction takes place in one half the time and execution speed is doubled. But physical laws limit the improvement that will be possible this way. Furthermore, as the physical limits are approached, improvements will become very costly. Fortunately there is another possibility for speeding up computations, parallelizing them.

Parallel Computing Hardware – Flynn's Classification

An early attempt to classify parallel computation made by Flynn is somewhat imprecise today but is nevertheless widely used.

	Single-data stream	Multiple-data streams
Single instruction	von Neumann	SIMD
Multiple instructions		MIMD

As we saw above, the original computer architecture, the von Neumann computer, reads instructions one at a time sequentially and acts upon data in the same way; thus, they are single instruction, single data, or SISD machines.

An early idea for parallelization, especially for scientific and engineering programming, has been the vector computer. Here it is often the case that the same instruction is performed on many data items as if these data were a single unit, a mathematical vector. For example, the scalar multiplication of a vector multiplies each component by the same number. Thus a single instruction is carried out on multiple data so these are SIMD machines. In these machines the parallelism is very structured and fine-grained (see Section 1.3).

Another term for this kind of computation is *data parallelism*. The parallelism stems from the data while the program itself is entirely serial. Mapping each instruction of the program to its target data is done by the compiler. Vector compilers automatically parallelize vector operations, provided the calculation is *vectorizable*, that is, can be correctly done in parallel (see Section 3.6).

Modern languages incorporate special instructions to help the compiler with the data partitioning. For example, the following statements in High Performance Fortran (HPF)

```
      real x(1000)
!HPF$ PROCESSORS p(10)
!HPF$ DISTRIBUTE x(BLOCK) ONTO p
```

invokes 10 processors and instructs the 1,000 elements of x to be distributed with $1{,}000/10 = 100$ contiguous elements going to each.

Another approach to SIMD/data partitioned computing, massively parallel SIMD, is exemplified by the now extinct Connection Machine. Here instructions are broadcast (electronically) to thousands of processors each of which is working on its own data.

True, flexible, parallel computation comes about with multiple independent processors executing, possibly different, instructions at the same time on different data, that is, multiple instruction multiple data or MIMD computers. This class is further categorized according to how the memory is configured with respect to the processors, centralized, and shared or distributed according to some topology.

We consider each of these in more detail below.

1.2 SIMD–Vector Computers

In the von Neumann model, much of the computer hardware is idle while other parts of the machine are working. Thus the Decode Unit is idle while the ALU is calculating an addition for example. The idea here is to keep all the hardware of the computer working all the time. This is parallelism at the hardware level.

Operation of a Vector Computer – Assembly-Line Processing

First the computer's hardware is modularized or *segmented* into functional units that perform well-defined specialized tasks (see, for example, the Cray architecture diagram Fig. 16). The vector pipes are likewise segmented. Figure 4 shows the segments for the Cray add pipe.

It is desirable that the individual units be as independent as possible. This idea is similar to the modularization of an assembly plant into stations each of which performs a very specific single task. Like a factory, the various detailed steps of processing done to the code and data of a program are formalized, and specialized hardware is designed to perform each such step at the same time as

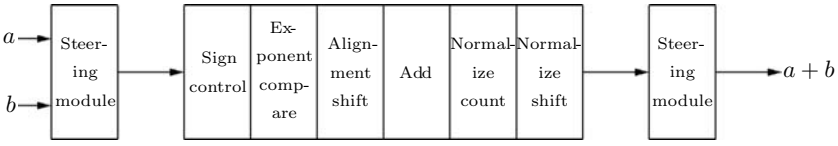

Fig. 4. A block diagram of the Cray add unit.

all the other steps. Then the data or code is processed step by step by moving from segment to segment; this is *pipelining*.

In our model of a computer, some of the main units are the fetch and store unit, the decode unit, and the arithmetic and logic unit. This makes it possible, for example, for the instructions of the program to be fetched before their turn in the execution sequence and held in special registers. This is called *caching*, allowing for advance decoding. In this way, operands can be prefetched so as to be available at the moment needed. Among the tasks of the decode unit is to precalculate the possible branches of conditional statements so that no matter which branch is taken, the right machine instruction is waiting in the instruction cache.

The innovation that gives a vector computer its name is the application of this principle to floating point numerical calculations. The result is an assembly line processing of much of the program's calculations. The assembly line in this case is called a *vector pipe*.

Assembly line processing is effective especially for the floating point operations of a program. Consider the sum of two vectors $\mathbf{x} + \mathbf{y}$ of length 200. To produce the first sum, $x_1 + y_1$, several machine cycles are required as we saw above. By analogy, the first item to roll off an assembly line takes the full time required for assembling one item. But immediately behind it is the second item and behind that the third and so on. In the same way, the second and subsequent sums $x_i + y_i$, $i = 2, \ldots, 200$, are produced one per clock cycle. In the next section we derive some equations governing such vector computations.

Example. Calculate $y_i = x_i + x_i^2$ for $i = 1, 2, \ldots, 100$

```
loop i = 1...100
    yi = xi*(1+xi)    or?    yi = xi + xi * xi
end loop
```

Not all operations on mathematical vectors can be done via the vector pipes. We regard a *vector operation* as one which can. Mathematically it is an operation on the components of a vector which also results in a vector. For example, vector addition $\mathbf{x} + \mathbf{y}$ as above. In components this is $z_i = x_i + y_i$, $i = 1, \ldots, n$, and would be coded as a loop with index i running from 1 to n. Multiplying two

Table 1. *Vector timing data*[*]

Type of arithmetic operation	Time in ns for n operations
Vector add/multiply/boolean	$1000 + 10n$
Vector division	$1600 + 70n$
Saxpy (cf. pp 14)	$1600 + 10n$
Scalar operation[**]	$100n$
Inner product	$2000 + 20n$
Square roots	$500n$

[*] For a mid-80's memory-to-memory vector computer.
[**] Except division, assume division is 7 times longer.

vectors componentwise and scalar multiplication, that is, the multiplication of the components of a vector by a constant, are other examples of a vector operation.

By contrast, the inner or dot product of two vectors is not a vector operation in this regard, because the requirement of summing the resulting componentwise products cannot be done using the vector pipes. (At least not directly, see the exercises for pseudo-vectorizing such an operaton.)

Hockney's Formulas

Let t_n be the time to calculate a vector operation on vectors of length of n. If s is the number of clock cycles to prepare the pipe and fetch the operands and l is the number of cycles to fill up the pipe, then $(s + l)\tau$ is the time for the first result to emerge from the pipe where τ is the time for a clock cycle. Thereafter, another result is produced per clock cycle, hence

$$t_n = (s + l + (n - 1))\tau,$$

see Table 1.

The *startup time* is $(s + l - 1)\tau$ in seconds. And the *operation rate*, r, is defined as the number of operations per unit time so

$$r = \frac{n}{t_n}.$$

Theoretical peak performance, r_∞, is one per clock cycle or

$$r_\infty = \frac{1}{\tau}.$$

Thus we can write

$$r = \frac{r_\infty}{1 + \frac{s+l-1}{n}}.$$

This relationship is shown in Fig. 5.

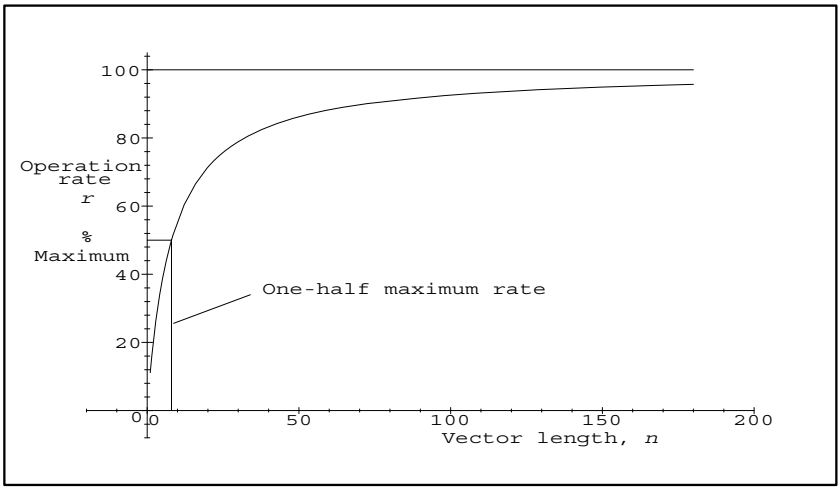

Fig. 5. Operation rate vs. vector length.

Hockney's $n_{1/2}$ value is defined as the vector length for achieving one-half peak performance, that is,

$$\frac{1}{2\tau} = \frac{n_{1/2}}{(s+l-1+n_{1/2})\tau}.$$

This gives

$$n_{1/2} = s + l - 1$$

or equal to the startup time. Using $n_{1/2}$, the operation rate can now be written

$$r = \frac{r_\infty}{1 + \frac{n_{1/2}}{n}}.$$

Hockney's *break-even point* is defined as the vector length for which the scalar calculation takes the same time as the vector calculation. Letting $r_{\infty,v}$ denote the peak performance in vector mode and $r_{\infty,s}$ the same in scalar mode, we have

$$\text{vector time for } n = \text{scalar time for } n$$
$$\frac{s+l-1+n_b}{r_{\infty,v}} = \frac{n_b}{r_{\infty,s}}.$$

Solving this for n_b gives

$$n_b = \frac{n_{1/2}}{\frac{r_{\infty,v}}{r_{\infty,s}} - 1}.$$