Appendix 3 Benchmarking

Benchmarking is a subject that receives both too little and too much attention. Too little, because knowing the relative speeds of machines, languages, and algorithms can have an enormous impact on your ability to obtain timely results. Too much, because tests that may have little bearing on practical problems can dominate manufacturers' advertising and users' purchase decisions.

Amid all of the hype, a simple recurring truth is that the best benchmark is a problem that you are interested in. An early standard was the *Linpack* set of subroutines, which have been run on an enormous range of machines (see the listing at `https://www.top500.org`). Because there is a great deal of specialized structure in these routines, some aggressive compilers started to have switches that recognized them and used carefully hand-tuned code to appear faster on this benchmark. To prevent that, as well as to cover a much broader range of applications, an industry-wide group has defined a suite of test problems called the *SPEC* benchmark (`https://www.spec.org`). This is a comprehensive set of programs covering many types of numerical algorithms. Where it's available, it's a reliable guide to machine speed. However, the suite may not be available for a particular machine that you're interested in, and it is not freely accessible. For this reason it's useful to have a simple test program that can provide a rough order-of-magnitude estimate of speed.

I've found it convenient to use a series expansion of π,

$$\begin{aligned}\pi &= 4\tan^{-1}(1)\\ &\approx \sum_{i=1}^{N} \frac{0.5}{(i-0.75)(i-0.25)} \quad .\end{aligned}$$

Summing this series requires five floating-point operations per step (ignoring the overhead for iterating the loop), providing an estimate of the computational speed by measuring the time taken to sum it. This is usually reported in the number floating-point operations per second, called *flops*. The total time should be linear in the number of terms used once N is large enough (there is always some overhead associated with starting and finishing execution). And since the correct answer is known, it is easy to check the validity and precision of the result.

Furthermore, it's instructive to write the series two ways, one as a reduction over a scalar term

$$\pi(N) = \sum_{i=1}^{N} \frac{0.5}{(i-0.75)(i-0.25)} \quad , \tag{A3.1}$$

Table A3.1. *Selected execution speeds to sum a series expansion of* π.

computer	*speed (Mflops)*	*program version*
IBM Blue Gene/P	1,595,914	C, MPI, 4096 processes
IBM Blue Gene/P	394	C
GTX 1080	1,088,907	CUDA, 3,906,250 blocks
Cray XT4	811,688	C, MPI, 2048 processes
Cray XT4	676	C
2.3 GHz Intel E5-2686	93,338	C, MPI, 32 processes
2.3 GHz Intel E5-2686	3,274	C, -03 -ffast-math
2.3 GHz Intel E5-2686	1,428	C
2.3 GHz Intel E5-2686	61,570	JavaScript, 32 workers
2.3 GHz Intel E5-2686	2,310	JavaScript
2.3 GHz Intel E5-2686	318	Python, NumPy
2.3 GHz Intel E5-2686	15.8	Python
Connection Machine CM-2	851	C, 32k processors
DEC XP1000	207	C
IBM ES/9000	148	C
IBM ES/9000	16.2	C, array
1 GHz Pentium III	134	C
Cray Y-MP4/464	118	C
Cray Y-MP4/464	10.0	C, array
DEC AlphaStation 500/500	70.0	C
HP 735	18.2	C
SUN Sparc 10/40	9.86	C
200 MHz Intel Pentium Pro	13.3	C
90 MHz Intel Pentium	6.33	C
Sun SPARCStation 2	4.50	C
Sun SPARCStation 1	1.21	C
DEC VAX 8650	1.72	C
25 MHz Intel 486	0.70	C
33 MHz Intel 386/387	0.35	C
25MHz 486/87	0.23	C
Sun 3/60 (68020)	0.036	C
12.5 MHz Intel 286	0.034	C
10 MHz Intel 8088	0.0011	C

and the other recursively in terms of an array

$$\pi(N) = \pi(N-1) + \frac{0.5}{(N-0.75)(N-0.25)} \quad . \tag{A3.2}$$

While these are formally equivalent, the extra storage required by the latter can inhibit schemes for caching and parallelization and hence probe the relative strengths of computer architectures and compilers.

Table A3.1 shows some sample speeds for machines, languages, and options. It is NOT in any way a thorough characterization of these systems, but it is an easily-generated estimate that is typically surprisingly close to much more careful benchmarks.

The single most remarkable feature of this table is that it spans roughly nine orders of magnitude. That's the difference between an algorithm taking about a minute and about the duration of recorded history. For some big problems, literally the fastest way to solve them has been to wait for a faster computer to be developed.