Appendix 2 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. Results for the *TOP500* systems are listed at `https://www.top500.org`, and since the largest ones use as much power as a whole city the *Green500* list (`https://www.top500.org/lists/green500`) tracks their relative efficiency.

Because there is a great deal of specialized structure in these routines, aggressive compilers used 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 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 system 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,

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

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*.

This is a classic parallel computation, a *scatter-gather* that can distribute terms in the sum to multiple processors and then collect the result. In particular, it is a type of *map-reduce*, in which each node calculates a value based on its index, and these are then centrally summed. There are more efficient ways to calculate pi [Bailey *et al.*, 1996], and because this is limited to the precision of the numbers used summing beyond

Table A2.1. *Selected execution speeds to sum a series expansion of π.*

speed (Gflops)	system	version
17,340,800	IBM AC922 (Summit)	C++, MPI, CUDA, 2048 nodes, 12228 GPUs
88,333	Cray XC40 (Theta)	C, MPI, OpenMP, 1024 nodes, 64 cores/node
16,239	NVIDIA A100	C++, CUDA, 8 GPUs
2,117	Intel 8175M	C, MPI, 10 nodes, 96 cores/node
2,102	Intel 8175M	Python, Numba, MPI, 10 nodes, 96 cores/node
2,052	NVIDIA A100	C++, CUDA, 6192 cores
1,595	IBM Blue Gene/P	C, MPI, 4096 processes
1,090	NVIDIA V100	Python, Numba, CUDA, 5120 cores
811	Cray XT4	C, MPI, 2048 processes
315	Intel 8175M	Python, Numba, 96 cores
267	Intel 8175M	C++, 96 threads
152	Intel 8175M	JavaScript, 96 workers
44.6	Intel i7-8700T	C, 6 threads
9.37	Intel i7-8700T	C, optimized
1.78	Raspberry Pi 4	C, 4 threads
0.85	Connection Machine CM-2	C, 32k processors
0.57	Intel i7-8700T	C, unoptimized
0.47	Intel i7-8700T	Python, NumPy
0.148	IBM ES/9000	C
0.134	Intel Pentium III	C
0.118	Cray Y-MP4	C, vector
0.074	Raspberry Pi Zero	C
0.029	Intel i7-8700T	Python
0.017	SAMD51J20A	C
0.013	Intel Pentium Pro	C
0.010	Cray Y-MP4	C, scalar
0.003	RP2040	Arduino
0.001	Sun SPARCStation 1	C
0.001	DEC VAX 8650	C
0.0007	Intel 486	C
0.0003	RP2040	MicroPython
0.0001	ATtiny1614	Arduino
0.00003	Sun 3/60	C
0.00003	Intel 286	C
0.000001	Intel 8088	C

that is wasted in round–off, but the calculation is simple to port to new systems, and the validity of the result is easy to check. Table A2.1 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 thirteen orders of magnitude! That's the difference between an algorithm taking the duration of recorded history and a fraction of a second. For some big problems, literally the fastest way to solve them was to wait for a faster computer to be developed.