Performance analysis

2017-08-24

Performance analysis is a rich area that combines experiment, theory, and engineering. In this class, we will explore all three. The purpose of this note is to set the stage; more specifically, we want to introduce

1. Performance analysis concepts that will recur throughout the course.
2. Common misconceptions and deceptions regarding performance

High performance computing (HPC), like most engineering disciplines, is about tradeoffs. The goal is to compute an answer that is good enough, fast enough, within the constraints of the available computing resources. But it is often not so simple to measure either solution quality or resource constraints.

1 What to measure?

The usual measure of performance in HPC is *time to solution*: the number of seconds of *wall clock time* to satisfy some termination criterion. It is important to specify wall clock time rather than CPU time, because fully utilizing all the CPUs in a large parallel machine is extremely challenging! In a *scaling study*, one attempts to quantify how the time to solution depends on the computing resources available, and possibly on the problem size or solution quality. To understand the results of a scaling study, it is often useful to look at measures derived from wall clock time, such as the effective flop rate, speedup, scaled speedup, or parallel efficiency. In addition to wall clock time, one might want to understand how much memory, disk space, bandwidth, or power is used in a given computation. But figuring out what to plot is a critical part of designing a performance study, and more exotic measures (flops/GW?) sometimes do more to obscure performance issues than to illuminate them.

Takeaway Think twice if you see anything but wall clock time to solution reported as a primary performance measure!

2 Basic performance concepts

Most performance studies involve at least two parameters:

- n – a measure of the problem size
- p – the number of processors

Understanding how run time varies these two parameters (and others) gives a general framework for understanding performance. A study in which p varies is called a *scaling study*, and is at the heart of most parallel performance analyses.

2.1 Algorithm complexity

The starting point for most performance analysis involves understanding how work scales with problem size. For example, computing the product of two $n \times n$ floating point matrices takes about $2n^3$ flops (floating point operations): n^3 adds and n^3 multiplies. In most introductory computer science classes, we focus use order notation to give a crude but concise description of how the work performed by an algorithm scales with problem size; that is, matrix multiply is $O(n^3)$. In an ideal world, we would say the time is proportional to n^3, and determine the proportionality constant by a single experiment. In practice, things are rarely so simple.

Simple asymptotic complexity models are absolutely the right place to *start* when thinking about work done by a computation, but they are only a starting point. The "hidden constant" in the order notation used in most complexity estimates can vary dramatically depending both on the nature of the algorithm and on details of how it is implemented – a well-tuned matrix multiplication routine can be orders of magnitude faster than a naive implementation on the same hardware. Moreover, in modern machines, the cost to communicate data can easily exceed the cost of floating point operations; an overly crude complexity estimate that does not account for this fact may yield wildly inaccurate predictions.

Takeaway Algorithm complexity tells us how work scales with problem size – useful, but not all there is to performance.

2.2 Response time and throughput

Some algorithms run to completion, then return a result more-or-less instantaneously at the end. In many cases, though, a computation will produce intermediate results. In this case, there are usually two quantities of interest in characterizing the performance:

- The time to first result (response time)
- Rate at which results are subsequently produced (throughput)

For problems involving information retrieval or communication, we usually refer to these as the *latency* and *bandwidth*. But the idea of an initial response time and subsequent throughput rate applies more broadly.

When we think about concepts of latency and throughput, we are measuring performance not by a single number (time to completion), but by a curve of utility versus time. When we think about a *fixed* latency and throughput, we are implicitly defining a piecewise linear model of utility versus time: there's an initial period of zero utility, followed by a period of linearly increasing utility with constant slope (until completion). The piecewise linear model is attractive in its simplicity, but more complex models are sometimes useful. For example, to understand the performance of an iterative solver, the right way to measure performance might be in terms of approximation error versus time.

Takeaway Time to *completion* is not always the right measure of progress.

2.3 Theoretical machine limits

The theoretical rate at which a code could run depends both on the characteristics of the code and on characteristics of the machine on which it runs. These include the *theoretical peak flop rate* the maximum rate at which floating point operations that could possibly be executed. Even this is not so straightforward to estimate, since modern processors are often capable of launching more than one floating point instruction in each cycle, which may be vector floating point instructions that can do several distinct floating point operations.

In addition to the computational capacity of the machine, it is important to understand something about the *latency* and *peak bandwidth* of any communication channels, I/O devices, and memory subsystems. For I/O and memory, one may also have to worry about the *peak capacity*; even if there is ample storage for a computation's data in main memory, it's hard to get good performance unless the *working set* that is most frequently accessed fits in relatively fast (and small) cache memories.

We will discuss some basics of machine architecture in the coming classes in order to set the context for understanding the theoretical limits of the machine. For now, we simply note that these limits are not so simple, and it is not simple to get to them: with tuning, a real code might make it to 10% of peak (depending on the nature of the computation).

2.4 Speedup, efficiency, and Amdahl's law

In a *strong scaling* study, one fixes the problem size n and studies performance as a function of p. The usual measure of scalability is the *speedup* of a parallel computation:

$$S(p) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(p)}$$

where T_{serial} is the performance of the *best* serial code available and $T_{\text{parallel}}(p)$ is the time required with p processors. The *parallel efficiency* is the ratio of the speedup to the ideal linear speedup:

$$\text{Efficiency}(p) = \frac{S(p)}{p} = \frac{T_{\text{serial}}}{pT_{\text{parallel}}(p)}.$$

A *speedup plot* of speedup versus p is a standard graphic in most of the HPC literature, and is simultaneously one of the most useful and one of the most abused plots around. We return to some of the issues with deceptive speedup plots later in these notes.

We rarely achieve ideal linear speedup (100% efficiency), in part because most real codes include some work that is difficult or impossible to parallelize. If α is the fraction of the serial work that cannot be parallelized, then *Amdahl's law* tells us the best scaling we can hope for is

$$S(p) \leq \frac{1}{\alpha + (1-\alpha)/p} \leq \frac{1}{\alpha}.$$

For example, if 10% of the work in a given computation is serial, we cannot hope for more than a 10× speedup no matter how many processors we use. In practice, this is usually a generous estimate: some overheads usually grow with the number of processors, so that past a certain number of processors the speedup often doesn't just level off, but actually *decreases*.

Takeaway Speedup quantifies scalability. For fixed problems, Amdahl's law tells us us that serial work limits the max speedup.

2.5 Scaled speedup, weak scaling, and Gustafson's law

In strong scaling studies, we assume we are interested in a fixed problem size n. In many cases, though, we are not interested in using ever-more parallelism to get faster answers to the same problems; rather, we want to use increased parallelism to solve bigger problems. In *weak scaling* studies, we usually consider the *scaled speedup*

$$S(p) = \frac{T_{\text{serial}}(n(p))}{T_{\text{parallel}}(n(p), p)}$$

where $n(p)$ is a family of problem sizes chosen so that the work per processor remains constant. For weak scaling studies, the analog of Amdahl's law is *Gustafson's law*; if a is the amount of serial work and b is the parallelizable work, then

$$S(p) \leq \frac{a + bP}{a + b} = p - \alpha(p - 1)$$

where $\alpha = a/(a + b)$ is the fraction of serial work.

Takeaway Fixing the problem size (strong scaling) is not the only way. Sometimes weak scaling studies are more instructive.

2.6 Pleasing parallelism and high throughput

A problem that can be decomposed into many independent tasks with little overhead used to be called *embarrassingly parallel*; these days, it is sometimes called *pleasingly parallel* instead. Monte Carlo simulations and many "big data analytics" tasks are pleasingly parallel. Because pleasingly parallel jobs have very low overheads associated with serial work or with synchronization,

they offer a high level of scalability, even on machines where communication and synchronization is expensive. Frameworks like Google's MapReduce thrive in part because there are indeed many embarassingly parallel computations in the world. At the same time, there are also many jobs that are not embarrassingly parallel.

The communities that deal primarily with embarassingly parallel jobs tend to be different than the traditional scientific HPC communities. In particular, where HPC usually focuses on time to completion (with some caveats noted above), communities focused on embarrassingly parallel tasks usually care about high *throughput*. If we sometimes refer to HTC (high-throughput computing) in this class, this is what we mean.

Takeaway Some tasks are easy to parallelize. Some are not. It helps to know the difference.

2.7 Theoretical and empirical performance models

> With four parameters, I can fit an elephant, and with five, I can make him wiggle his trunk.
> – Von Neumman

A *performance model* predicts the performance of some code as a function of problem size, parallelism, and perhaps other parameters. Performance models are useful to the extent that they help us predict whether we can meet a performance goal and to the extent that they can guide us to where our codes most need improvement (or where they will run into scaling bottlenecks on future machines). Because models reflect our understanding, they may be incomplete; indeed, the most useful models are *necessarily* incomplete, since otherwise they are too cumbersome to reason about! Experiments reflect what really happens, and are a critical counterpoint to models.

The division between performance models and experiments is not sharp. In the extreme case, machine learning and other empirical function fitting methods can be used to estimate how performance depends on different parameters under very weak assumptions. When strongly empirical models have many parameters, a lot of data is needed to fit them well; otherwise, the models may be *overfit*, and do a poor job of predicting performance except away from the training data. This may be appropriate for cases where the model is used as the basis for *auto-tuning* a commonly-used kernel for a

particular machine architecture, for example. But performance experiments often aren't cheap – or at least they aren't cheap in the regime where people most care about performance – and so a simple, theory-grounded model is often preferable. There's an art to balancing what should be modeled and what should be treated semi-empirically, in performance analysis as in the rest of science and engineering.

Takeaway Both theory and experiment are needed for performance modeling.

2.8 Applications, benchmarks, and kernels

The performance of application codes is usually what we really care about. But application performance is generally complicated. The main computation may involve alternating phases, each complex in its own right, in addition to time to load data, initialize any data structures, and post-process results. Because there are so many moving parts, its also hard to use measurements of the end-to-end performance of a given code on a given machine to infer anything about the speed expected of other codes. Sometimes it's hard even to tell how the same code will run on a different machine!

Benchmark codes serve to provide a uniform way to compare the performance of different machines on "characteristic" workloads. Usually benchmark codes are simplified versions of real applications (or of the computationally expensive parts of real applications); examples include the NAS parallel benchmarks, the Graph 500 benchmarks, and (on a different tack) Sandia's Mantevo package of mini-applications.

Kernels are frequently-used subroutines such as matrix multiply, FFT, breadth-first search, etc. Because they are building blocks for so many higher-level codes, we care about kernel performance a lot; and a kernel typically involves a (relatively) simple computation. A common first project in parallel computing classes is to time and tune a matrix multiplication kernel.

Parallel *patterns* (or "dwarfs") are abstract types of computation (like dense linear algebra or graph analysis) that are higher level than kernels and more abstract than benchmarks. Unlike a kernel or a benchmark, a pattern is too abstract to benchmark. On the other hand, benchmarks can elucidate the performance issues that occur on a given machine with a particular type of computation.

Takeaway Application performance is complicated. We try to simplify by looking at benchmark codes and kernels, or by understanding the performance characteristics of common computational patterns.

3 Designing performance experiments

3.1 Timing and profiling

Profiling involves running a code and measuring how much time (and resources) are used in different parts of the code. For codes that show any data-dependent performance, it is important to profile on something realistic, as the time breakdown will depend on the use case. One can profile with different levels of detail. The simplest case often involves manually instrumenting a code with timers. There are also tools that *automatically instrument* either the source code or binary objects to record timing information. *Sampling profilers* work differently; they use system facilities to interrupt the program execution periodically and measure where the code is. It is also possible to use *hardware counters* to estimate the number of performance-relevant events (such as cache misses or flops) that have occurred in a given period of time. We will discuss these tools in more detail as we get into the class (and we'll use some of them on our codes).

As with everything else, there are tradeoffs in running a profiler: methods that provide fine-grained information can produce a *lot* of data, enough that storing and processing profiling data can itself be a challenge. There is also an issue that very fine-grained measurement can interfere with the software being measured. It is often helpful to start with a relatively crude, lightweight profiling technology in order to first find what's interesting, then focus on the interesting bits for more detailed experimentation.

Profiling has different goals. The most common reason to profile is to find *performance bottlenecks*: in many codes, the majority of the time is spent in one or a few pieces of a large code base, and it makes sense to find where the time is spent in order to spend tuning time in a sensible way. Good compilers can also use profile data as the basis of *profile-directed optimization*.

3.2 Experimental issues

Compared to most experimental sciences, computer scientists have it easy: our only safety issues involve RSI, and if an experiment breaks, we can just re-run it with little additional work. But, as with other experimental work, performance analysis does require that we understand issues that lead to variation in results. For example:

- When running the same computation twice (in the same process), the second run will often be faster because the first run "warms the cache." We will discuss this in more detail when we discuss computer architecture basics.
- Two codes running on the same machine can interfere with each other (e.g. by stealing memory bandwidth or thrashing shared caches), even if they are nominally not trying to use the same cores.
- Timers have finite resolution, and so it may be necessary to run a short code segment repeatedly in order to get use enough time to get an accurate measurement.

All this suggests that it is important to understand the limitations of experimental measurement, and also the environmental factors that cause variations in measurements. We will return to these issues periodically throughout the class.

4 Engineering for performance

Performance models and experiments help us both to design new codes for high performance and to tune the performance of existing codes. We will spend a lot of time talking about engineering for performance over the course of the semester, but let's take a moment now to talk about a few recurring themes.

4.1 Know when to tune

There are many reasons *not* to tune code:

- Tuning takes human time. If the human time is more expensive than the computation time saved, it's not worth it.

- Performance is often in tension with maintainability, generality, and other nice software design properties. If tuning for performance means making a mess of the code base, it may not be worth it.
- Most codes have bottlenecks where the majority of the time is spent. It doesn't make sense to tune something that already takes little time.

We will mostly focus in the class on *what* to tune, but in general it is worth asking also *why* a code should be tuned.

4.2 Tune data structures

On modern machines, memory access and communication patterns are critical to performance. Because of this, tuning often involves looking not at *code* but at the *data* that the code manipulates. Many of the optimizations we will discuss in this class involve data structures: rearranging arrays for unit stride, simplifying structures with many levels of indirection, or using single precision for high-volume floating point data, etc. With a proper interface abstraction, the fastest way to high performance often involves replacing a low-performance data structure with an equivalent high-performance structure.

4.3 Expose parallelism

Achieving good performance on modern machines is increasingly about making good use of parallel computing resources. This means not only explicit parallelism (using threads or MPI, for example), but also taking advantage of the parallelism available inside a single core. A lot of the class will involve figuring out where there are opportunities for parallelism, and exposing those opportunities to compilers, frameworks, or ourselves!

4.4 Use the right tools

Performance tuning is hard. We make our lives easier by using the best tools we can get our hands on: good compilers, well-tuned libraries and frameworks, profilers and performance visualization tools, etc. And when the right tools don't exist, sometimes we get to make them ourselves!

5 Misconceptions and deceptions

> It ain't ignorance causes so much trouble; it's folks knowing so much that ain't so.
> – Josh Billings

One of the common findings in pedagogy research is that an important part of learning an area is overcoming common *misconceptions* about the area; see, e.g. [6], [7], [8]. And there are certainly some common misconceptions about high performance computing! Some misconceptions are exacerbated by bad reporting, which leads to deceptions and delusions about performance.

5.1 Incorrect mental models

Algorithm = implementation We can never time algorithms. We only time implementations, and implementations vary in their performance. In some cases, implementations may vary by orders of magnitude in their performance!

Asymptotic cost is always what matters We can't time algorithms, but we can reason about their asymptotic complexity. When it comes to scaling for large n, the asymptotic complexity can matter a lot. But comparing the asymptotic complexity of two algorithms for modest n often doesn't make sense! QuickSort may not always be the fastest algorithm for sorting a list with ten elements...

Simple serial execution Hardware designers go to great length to present us with the *interface* that modern processor cores execute a instructions sequentially. But this *interface* is not the actual *implementation*. Behind the scenes, a simple stream of x86 instructions may be chopped up into micro-instructions, scheduled onto different functional units acting in parallel, and executed out of order. The *effective behavior* is supposed to be consistent with sequential execution – at least, that's what happens on one core – but that illusion of sequential execution does not extend to performance.

Flops are all that count Data transfers from memory to the processor are often more expensive than the computations that are run on that data.

Flop rates are what matter What matters is time to solution. Often, the algorithms that get the best flop rates are not the most asymptotically efficient methods; as a consequence, a code that uses the hardware less efficiently (in terms of flop rate) may still give the quickest time to solution.

All speedup is linear See the comments above about Amdahl's law and Gustafson's law. We rarely achieve linear speedup outside the world of embarrassingly parallel applications.

All applications are equivalent Performance depends on the nature of the computation, the nature of the implementation, and the nature of the hardware. Extrapolating performance from one computational style, implementation, or hardware platform to another is something that must be done very carefully.

6 Deceptions and self-deceptions

In 1991, David Bailey wrote an article on "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers". It's still worth reading (as are various follow-up pieces – see the Further Reading section), and highlights issues that we still see now. To summarize slightly, here's my version of the list of common performance deceptions:

6.1 Unfair comparisons and strawmen

A common sin in scaling studies is to compare the performance of a parallel code on p processors against the performance of the *same code* with $p = 1$. This ignores the fact that the parallel code may have irrelevant overheads, or (worse) that there may be a better organization for a single processor. Consequently, the speedups no longer reflect the reasonable expectation of the reader that this is the type of performance improvement they might see when going to a good parallel implementation from a *good* serial implementation. Of course, it's also possible to see great speedups by comparing a bad serial implementation to a correspondingly bad *parallel* implementation: a lot of unnecessary work can hide overheads.

A similar issue arises when computing with accelerators. Enthusiasts of GPU-accelerated codes often claim order of magnitude (or greater) per-

formance improvements over using a CPU alone. Often, this comes from explicitly tuning the GPU code and not the CPU code. A 2010 paper out of Georgia Tech gives several examples where, after tuning, two quad-core CPU sockets gave roughly the same performance as one or two GPUs.

6.2 Using the wrong measures

If what you care about is time to solution, you might not really care so much about watts per GFlop (though you certainly do if you're responsible for supplying power for an HPC installation). More subtlely, you don't necessarily care about scaled speedup if the natural problem size is fixed (e.g. in some graph processing applications).

6.3 Deceptive plotting

There are so many ways this can happen:

- Use of a log scale when one ought to have a linear scale, and vice-versa;
- Choosing an inappropriately small range to exaggerate performance differences between near-equivalent options;
- Not marking data points clearly, so that there is no visual difference between data falling on a straight line because it closely follows a trend and data falling on a straight line because there are two points.
- Hiding poor scalability by plotting absolute time vs numbers of processors so that nobody can easily see that the time for 100 processors (a small bar relative to the single-processor time) is equivalent to the time for 200 processors.
- And more!

Plots allow readers to absorb trends very quickly, but it also makes it easy to give wrong impressions.

6.4 Too much faith in models

Any model has limits of validity, and extrapolating outside those limits leads to nonsense. Treat with due skepticism claims that – according to some model – a code will run an order of magnitude faster in an environment where it has not yet been run.

6.5 Undisclosed tweaks

There are many ways to improve performance. Sometimes, better hardware does it; sometimes, better tuned code; sometimes, algorithmic improvements. Claiming that a jump in performance comes from a new algorithm without acknowledging differences in the level of tuning effort, or acknowledging non-algorithmic changes (e.g. moving from double precision to single precision) is deceptive, but sadly common. Hiding tweaks in the fine print in the hopes that the reader is skimming doesn't make this any less deceptive!

7 Rules for presenting performance results

In the intro to the book Performance Tuning of Scientific Applications, David Bailey suggests nine guidelines for presenting performance results without misleading the reader. Paraphrasing only slightly, these are:

1. Follow rules on benchmarks
2. Only present actual performance, not extrapolations
3. Compare based on comparable levels of tuning
4. Compare wall clock times (not flop rates)
5. Compute performance rates from consistent operation counts based on the best serial codes.
6. Speedup should compare to best serial version. Scaled speedup plots should be clearly labeled and explained.
7. Fully disclose information affecting performance: 32/64 bit, use of assembly, timing of a subsystem rather than the full system, etc.

8. Don't deceive skimmers. Take care not to make graphics, figures, and abstracts misleading, even in isolation.

9. Report enough information to allow others to reproduce the results. If possible, this should include

   - The hardware, software and system environment
   - The language, algorithms, data types, and coding techniques used
   - The nature and extent of tuning
   - The basis for timings, flop counts, and speedup computations

8 Questions

1. A given program spends 10% of its time in an initial startup phase, and then 90% of its time in work that can be easily parallelized. Assuming a machine with homogeneous cores, plot the idealized speedup and parallel efficiency of the overall code according to Amdahl's law for up to 128 cores. If you know how, you should use a script to produce this plot, with both the serial fraction and the maximum number of cores as parameters.

2. Suppose a particular program can be partitioned into perfectly independent tasks, each of which takes time τ. Tasks are set up, scheduled, and communicated to p workers at a (serial) central server; this takes an overhead time α per task. What is the theoretically achievable throughput (tasks/time)?

3. Under what circumstances is it best to not tune?

4. The class cluster consists of eight nodes and fifteen Xeon Phi accelerator boards (details under the "Computing platform" section of the syllabus). Based on an online search for information on these systems, what do you think is the theoretical peak flop rate (double-precision floating point operations per second)? Show how you computed this, and give URLs for where you got the parameters in your calculation. (We will return to this question again after we cover some computer architecture.)

5. What is the theoretical peak flop rate for your own machine?

9 Further reading

1. David Bailey, Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers.
2. David Bailey, Misleading Performance Reporting in the Supercomputing Field.
3. David Bailey, Twelve Ways to Fool the Masses: Fast Forward to 2011.
4. George Hager, Modern "Fooling the Masses" stunts blog posts.
5. David Bailey, Robert Lucas, and Samuel Williams (eds), Performance Tuning of Scientific Applications.
6. Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat (Efe) Guney, and Aashay Shringarpure. On the Limits of GPU Acceleration.