Week 1: Monday, Jan 25

1 Course mechanics

1. Office hours: TW 2-3 in Upson 5137 (or by appointment)
2. Grading:
   (a) 10% individual assignments
   (b) 60% group programming assignments
   (c) 30% final project (also a group)
3. Prerequisites:
   (a) C programming language and basic coding savvy
   (b) Some familiarity with numerical mathematics

   If you have some experience with these topics but not a lot, no worries – part of the rationale for doing most of the work in groups is that some of you will have more programming experience and others will have more background in numerical computing. Note that you must have a *reading* knowledge of C in order to work with code that I will give you, but you may also write code in Fortran if you prefer.
4. Rough topic outline:
   (a) Basic architecture and serial performance tuning
   (b) Parallel programming models and machines
   (c) Parallel programming with OpenMP, MPI, and UPC (?)
   (d) Patterns in parallel program organization. Collela's "7 dwarves":
      i. Computation on structured grids
      ii. Computation on unstructured grids
      iii. Spectral methods and FFTs
      iv. Dense linear algebra
      v. Sparse linear algebra
      vi. Particle methods

vii. Monte Carlo / embarrassingly parallel apps

(e) General techniques.

i. Ideas about work partitioning and load balancing.

ii. Tools for performance measurement and tuning.

iii. Building on and around libraries.

(f) Application examples

5. No text — notes and slides. Will post reference links as we go.

2 What are we about?

Our goal in this class is to learn how to solve big scientific problems fast. The words "big" and "fast" are deliberately vague, since every increase in computing power seems to bring about a corresponding increase in computing ambition. Remember the infamous Bill Gates quote "640K should be enough for anybody?"

Roughly speaking, solving scientific problems with computers involves three different types of activity:

1. *Application modeling* is about building quantitative models of the world that (a) are amenable to computation and (b) actually give useful information about some problem of interest.

2. *Mathematical analysis* is critical to scientific computations, both because mathematics is the language in which the models are posed and because of the mathematics needed to devise fast, stable algorithms that solve those ideas. The fastest algorithms for are often those that take the most advantage of mathematical structure of a specific class of problems.

3. *Software engineering* is the process of building fast, robust, re-usable, well-documented programs and libraries. If we're going to solve problems on the computer, it helps to think about how to write software to use both our time and the computer time efficiently.

There are performance implications in each of these activities, even at the application modeling level. We will try to give a broad picture of performance

that includes these types of modeling tradeoffs, though we will naturally emphasize numerical methods and software design.

There are many areas in which parallel computers are widely used for analyzing models; an incomplete list might include

1. Climate modeling
2. Astrodynamics
3. Computational biology (e.g. protein folding and company)
4. Computational chemistry
5. Computational material science and material design
6. Semiconductor device design
7. Structural modeling (e.g. building design)
8. Computational fluid dynamics (e.g. simulated wind tunnels)
9. Combustion modeling
10. Crash simulation (for cars or other things)
11. Financial modeling
12. Cryptography (NSA keeps Cray afloat)
13. Data mining, social network analysis, ...?

The scale of many of these computations is daunting. For example, consider climate modeling. The earth's surface is about half a billion square kilometers, so even a model with grid cells 50 km on a side will give us about half a million grid locations on the earth's surface. To get reasonable resolution of the atmosphere, we might need to model tens of layers vertically, giving us a few million cells in all; and in each cell we might have many variable describing wind speeds, temperature, pressure, moisture levels, and atmospheric chemistry. Thus even a coarse model of the climate will involve *at least* millions of unknowns, and probably more. Considering that these

quantities evolve over time, the task of solving realistic atmospheric models might reasonably be called gargantuan.[1]

As another example, consider transistor-level simulation of a modern computer chip. A modern Pentium has on the order of 800 million transistors! Again, that's a lot of computation. Of course, there are other levels of modeling, too. A coarser simulation, at the level of gates or above, would suffice for much of the testing, while computing parasitic capacitances and cross-talk between wires in the interconnect requires electromagnetic field solvers.

How do we get the amount of computing power to solve problems with many millions of unknowns? There are a few (not mutually exclusive) strategies:

- Choose algorithms carefully, or come up with a new algorithm. This is often not straightforward, but is usually important.
- Carefully tune our codes so that they run as fast as possible on each processor. We'll discuss this in some detail in the next couple weeks.
- Wait for 18 months and watch the performance double. This *used* to be a viable strategy, but is far less so now.
- Use multiple processors in parallel. This, of course, will be the major topic of the course.

3 Why parallel?

In 1965, Gordon Moore wrote an article in which he observed that the number of transistors on a chip doubles every 18 months. Remarkably, this trend *still* appears to hold, though at some point we will run into physical limits. It used to be that the combined effect of increased circuit density, increased clock rates, and ever more clever architectural features brought about roughly the same doubling in the serial performance of roughly equivalent machines, again every 18 months. As of a couple years ago, though, this latter trend stopped, for several reasons:

[1] Of course, behavior of the climate is also determined by the dynamics of oceans, ice sheets, etc. The level of resolution demanded to model each of these probably varies.

1. With larger numbers of components comes lower yield (i.e. a smaller fraction of the chips that are fabricated actually work correctly). Estimates of yields on the first generation of Cell processors (which power the PlayStation 3), for example, were 10%-20%! These processors have eight "processing elements," but Sony ships with one turned off in order to improve yield.

2. The power density on the Pentium 4 was 46 W/cm^2. If we had continued that trend, we would have quickly ended up with power densities on the order of a rocket nozzle (a couple hundred W/cm^2). These high power densities mean higher temperatures, which increase the resistance and lower the currents (and thus speeds) – and, in a nasty feedback loop, leakage current increases self heating. Apart from problems with failing circuits, high temperatures cause problems for packaging, which may be built from materials with different thermal coefficients of expansion. And without cooling, modern chips would be hotter than the surface of the sun (about 6000 C)!

3. Aside from high power densities, there is an issue with the absolute amount of power that a CPU burns. In part, this is driven by mobile devices — longer battery lifetimes for your laptop come only if you have higher better batteries or lower power requirements — but it is an issue for big computer users, too. Power-efficient supercomputing is a hot topic at the moment (pun intended).

4. For a long time, we were getting better performance through automatic instruction-level parallelism (ILP), which we'll discuss more in the next lecture or two — but that, too, has petered out.

Apart from these (very strong) engineering reasons, there are some physical limitations that prevent us from using serial computers for arbitrarily large computations. The speed of light in vacuum is 3×10^8 meters/second, or 0.3 meters/nanosecond, so one light-nanosecond is about a food. One light-clock cycle on a modern 3GHz machine, therefore, would be about 10 cm or 4 in. On a machine theoretically capable of running at THz frequencies (10^{12} cycles/second), a light-clock cycle would be 0.3 mm! If one wanted to also make a terabyte of memory accessible within one clock cycle at that rate, each bit could only take about two square angstroms — the size of a not-very-large atom. Clearly something has to give.

As a related calculation, consider that the size of a big supercomputer is on the order of magnitude of a football field — about half a light microsecond. Even in the best case, it *must* take thousands of cycles to get a message from one end of such a machine to the other, just because of the speed of light!

Because it seems impractical to continue speeding up serial computers indefinitely, parallelism becoming the norm, even in ordinry desktops and laptops. Consider my laptop, a 2.5 GHz Macbook Pro that I got about a year ago:

- This machine is dual core, and more cores/chip will probably be the norm in the future.
- Each individual core has implicit ILP and some short vector units (more on this later)
- The machine has a highly parallel NVidia graphics processing unit. If I get around to updating to the newest OS X (Snow Leopard) release, I will automatically get OpenCL, a parallel programming system built to take advantage of such hardware. As it is, I have installed the CUDA system from NVidia, which is also used to perform general-purpose computations on the GPU.
- And I spend an enormous amount of time using my laptop to connect to Google, which harnesses a ridiculous number of machines... and now you can harness the "cloud" too?

4 Parallel ideas

Rarely is it straightforward to make a program that runs for 1000 hours on one processor run for 1 hour on a machine with 1000 processors. Keeping many parallel processors usefully engaged is tricky. In part, this is because some parts of a program — such as handing out work to each processor or accumulting partial results into a single final result – are not easily parallelized. If we have a program in which some fraction s of the work is serial, then *Amdahl's laws* says that running on P processors can give speed things up by a factor of at most

$$\text{speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \leq \frac{1}{s + (1-s)/P} \leq \frac{1}{s}.$$

In order to get a thousand-fold speedup on *any* number of processors, then, at most 0.1% of our program time can be spent on serial work. Thus, the first challenge of parallel programming is *finding enough parallelism*.

Another challenge in parallel programming is *managing overheads* such as the time taken to start processes, communicate data, synchronize processes, or perform redundant computations. These overheads sometimes make it *slower* to run a program on many processors than it would be to run on just one or a few processors.[2]

Ideally, we would like to amortize parallel overheads; that is, we would like to spend a lot of time doing useful work relative to the time that we waste on overhead. Consequently, *granularity* is a big issue in parallel program design: we want each processor to be able to do a lot of work without communicating (coarse grain parallelism) in order to hide communication overhead, but we want to be able to handle small chunks of work (fine-grain parallelism) so that we can use a lot of processors.

Because communication between processors is expensive, *locality* is key: ideally, each processor should be able to compute with local information without communicating too frequently with other processors. Moreover, locality is key even to serial performance on modern machines! The memory in a typical machine is organized in a hierarchy, with fast (but small) *cache* memory close to the chip and slower (but bigger) storage further away. A typical modern machine has a tiny amount of memory in registers that can be accessed instantly; 32KB or 64KB of memory in level 1 cache that can be accessed in a cycle or two; a couple MB that can be accessed in around 10 cycles; and main memory that takes 50 or 100 cycles to access. Data on disk, or on other machines, takes thousands of cycles to access.

Why does it take so long to fetch data? In part, this is because DRAM latency (the time between when a request for data is initiated and when the first pieces of data are returned) has been improving much more slowly than processor speeds — 7% per year versus 60% per year. As for communication with remote nodes, remember the calculation from earlier: light only travels about four inches in a clock cycle.

Load balance is also crucial to machine performance. The speed of a parallel computation is limited by the slowest participant; if one processor

[2] An unfortunate cheat in measuring the performance of parallel codes is to compare the performance of a parallel program not to that of a corresponding serial code, but to that of the same parallel program, with all its overheads, running on a single node.

does the lion's share of the work, then having lots of processors doesn't help much. One processor can end up with a relatively heavy load in a heterogeneous computation (e.g. if all of the shock front in an explosion is simulated on one or a few processors), or in situations where there is insufficient parallelism (e.g. if one processor ends up doing serial work while all the other processors twiddle their virtual thumbs three billion times per second).

Given the difficulty of making full use of a parallel machine, how fast can we really go? Almost nobody achieves even near the theoretical machine peak speed, the rate of calculation possible if no coordination or memory access were necessary. It's even hard to reach the speed of a benchmark code like the LINPACK benchmark (the "hello world" of parallel computing). In fact, getting 10% of peak performance can be pretty good!

5 Thinking about performance

If the goal is the fastest time to a useful solution, the rate at which arithmetic is performed (floating point operations per second, or flop/s) is a terrible measure of performance! Before thinking about parallelizing a code, or tuning an existing code, it is worth pondering a few questions:

1. Are you using the right model for the job? Would it make sense to use a higher-level model – using SPICE rather than solving Maxwell's equations, or modeling a structure in terms of beams and plates rather than via a general 3D finite element discretization?

2. Are you using the right algorithm for the job? Note that often the fastest algorithms have relatively poor performance in terms of flop/s: it's easier to parallelize a (slow) Jacobi solver for elliptic PDEs than to parallelize a (faster) multigrid solver, and it's easier to parallelize the naive $O(n^2)$ computation of electrostatic forces between n particles than it is to parllelize the $O(n)$ fast multipole method.

3. Is it worth the programmer time to tune? It's not worth a day of programming to cut an hour off the runtime of a job that will run only once... though "one-off" jobs have a tendency to live longer than you expect. *Libraries* are key here.

Once we have decided on a model and a solution method, there is a basic procedure to getting good performance:

1. Measure the code in order to understand performance (profiling)
2. Find bottlenecks
3. Get rid of bottlenecks
4. Repeat

Simple performance models are also very helpful.

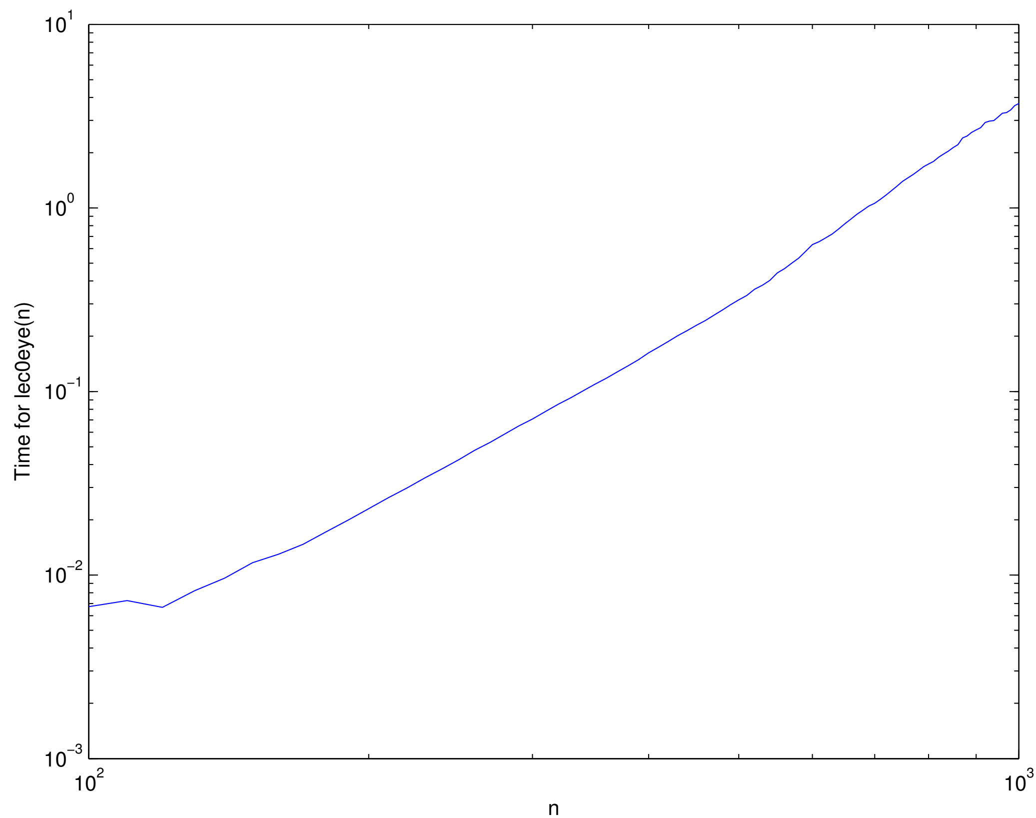
Measuring and understanding performance is tricky! We need to address questions of what constitutes a "fair" benchmark, how scaling should be mesured, etc. Ultimately, we would like experiments that help us arrive at a mental model that we can use to guide our efforts. This is important because the naive mental model that most of us use is often wrong! For example, consider the following MATLAB code fragment:

```
function I = lec0eye(n)
% Compute an n-by-n identity mtrix

  I = [];
  for k = 1:n
    I(k,k) = 1;
  end
```

On my laptop, this routine takes 3.7 seconds to run when $n = 1000$. A log-log plot of the run time versus n (Figure 5) shows that the time grows by nearly a factor of 1000 when n grows by a factor of 10 — that is, the observed cost is $O(n^3)$, though the programmer writing the code[3] probably thought that it ran in $O(n)$ time. The problem, of course, is that at step k of the loop, MATLAB expands the matrix by allocating a new chunk of memory and copying $O(k^2)$ elements from the old memory, for an overall cost of $\sum_{k=1}^{n} O(k^2) = O(n^3)$ work. But that copy is not immediately obvious in the code, and so *the most expensive part of the routine is missing from the programmer's mental model.* The programmer might be able to see what went wrong by using MATLAB's profiler (type `help profile` in MATLAB

[3]Said programmer is a figment of my imagination in this example, but I've seen the same performance error more than once in other settings.



to learn more). Even with the help of the profiler to see *where* the problem occurs, though, a novice programmer might not understand enough about how MATLAB works to understand *why* the problem occurs.

Parallel performance builds on serial performance, and our goal in the next couple lectures is to understand enough about the elements of modern computer architecture to understand *why* serial codes might be slow — and what we can do to speed them up.