Parallel II

Ben Klemens

13 March 2009

In the last episode, I pointed out that every year we get more, smaller transistors, but it's harder and harder to invent new tricks to string them together to do math faster.

So what's the easy solution? Instead of ramping up the speed of a processor, just get two. Thus, the advertised speed gains in the last several years have been via including two semi-distinct processors in one package—the multi-core chip.

On a larger scale, it's the same story: much of large-scale computing is really parallel computing. Systems like the NIH's Biowulf cluster are a pile of a few hundred separate computers, interlinked with high-speed networking cable.

Unfortunately, having two processors does not magically make your programs run twice as fast. Some work is fundamentally sequential, and can't be split into simultaneously-running subparts. To give a somewhat time-of-year appropriate example, think about filling out federal tax forms. Some parts can be filled out without regard to others: if you were a multi-core processor, core one could be filling in the name/address part, core two could be gathering data for the itemized deduction form, and core three could be totalling up sources of income, all at the same time and independent of each other. But you can't calculate total taxes owed until you've worked out all your income and deductions. The tax owed process on core four will be sitting around on its little transistor hands until the others are done.

Then, once all that's done, you can do your state taxes.

So even for a process as easy and simple as doing your taxes, you've already got some parts that can run in parallel and some that have to be run in sequence (i.e., in serial). Without looking at the hairy details, it's hard to say whether having two cores will double the speed of a program or do absolutely nothing.

So that's the central problem for today's story. If we want next year's program to run faster than this year's, we can't just wait for Intel or AMD to ramp up their MIPS count, because, as per in entry #002, they aren't really playing that game anymore. The cheapest expansion is now adding capacity to run parallel processes.

So, how is this transition to parallel computing going to work?

From here, we can work at a few levels, some of which are more amenable to parallelization than others. At the overall operating system level, for example, things are smooth and easy: put the drawing program on one core, the spreadsheet on another core, the general OS on another, and you can expect that these separate programs can do most of their work independently.

So for those of you who tend to have a dozen programs up at once (that's me), or for those of you using a multi-user system where security dictates that programs generally

Figure 1: Parallel threads let you do more, but there can still be bottlenecks.

shouldn't be talking to each other at all, more cores equals less gridlock.

Computing platforms But scientific computing tends to be about a single resource-intensive algorithm. This breaks down into two sub-threads: how well does the single program you're using thread itself, and how well does your specific algorithm thread?

First, the platform. There are many anecdotes like the one about Lotus and Microsoft at the top of the linked page, where a company succeeded by writing software that was bloated for its time, but two years later ran great. But that was the mid-90s, which is the steepest range in Figure 1 from last episode.

You're not writing raw assembly code, which means that you're using some sort of platform sitting between you and the cores: maybe a spreadsheet, or a stats package like Matlab, or a programming language like Python or C. The odds are good that the platform was originally written in the good ol' days that Joel the guru was talking about, back when parallelization wasn't a consideration, nor was code that worked a processor a little too hard.

How easy your life will be depends on the implementers of your platform, because it's up to them to correctly thread the internals where possible, and hand you tools to thread your own work. This one is a minefield, and some platforms have much better threading, both internal and user-side, than others. I won't name names.

How does the package that's my fault, Apophenia, fare on threadability? It does OK. The `apply` and `map` family of functions will look at the `apop_opts.thread_count` variable, and break the process down into the appropriate number of independent threads. So in a situation where you have a matrix or vector of a million rows, each of which can be independently processed, such as calculating a log likelihood, you can just set a single variable and go. If things get more complex than that, you may have to start rewriting your code. Given the tax code example above, this is not so surprising: I as the package author can't guess what sort of interdependencies your system may have. But I'll admit that there are more cases that could be handled on the back end.

Stats methods At the algorithmic level, some types of research do break into parallel threads more easily, the most obvious being agent-based or simulation-type methods. Here, you have a few million agents or particles, each of which has set rules for interacting with other agents. The core of the simulation, for each period, is a loop that updates the status of each agent. For most situations, with four cores, you can touch base with the first quarter of your agents on the first core, the second quarter on the second, and so on.

Also easy: drawing random numbers. If I want to create a posterior distribution via Bayesian updating, I can basically make independent draws from the prior and independently use them to grow the posterior. [But remember that each thread will need its own seed for the RNG. The easiest way to do this is to simply give thread zero a seed of 0, thread one a seed of 1, and so on.]

We thus have a specific way in which technology affects how we do research: agent-based models are going to be cheaper every year, as are other methods that feel out a model via random draws.

We assume that many data sets [but by no means all!] are produced via a method that makes each independent of the other. The word 'independent' means that massive data sets [e.g., over on the other screen I have a search using 1.48 billion data points] are easily parallelized, since processing on one point is independent of processing on the others.

Other methods write down a fixed model and search for the optimum. These usually involve stepping along and trying a sequence of new points. Is this candidate better than the current? If so, then we'll use that point until a better one comes along. This is inherently sequential, so the options for parallelization are much more limited.

Summary paragraph So the future is in parallelization, which is a somewhat different game from the game we had in the go-faster '90s, and that means the winners in the future may not be the winners today. Many programs (I'm still not naming names) have internals that don't thread well—lots of important global variables, lots of bottlenecks—and those will seem slower because they can't farm work out to multiple processors. Even statistical methods that were popular a decade ago may fall beind relative to methods like agent-based simulations and those that use random draws to feel out a distribution.

I'll have a little bit more on this in a few episodes, but I promised you guys some stats, so next episode I'll introduce the great statistical war.