COMPUTING / SOFTWARE

FEATURE

The Trouble With Multicore

Chipmakers are busy designing microprocessors that most programmers can't handle

By DAVID PATTERSON / JULY 2010

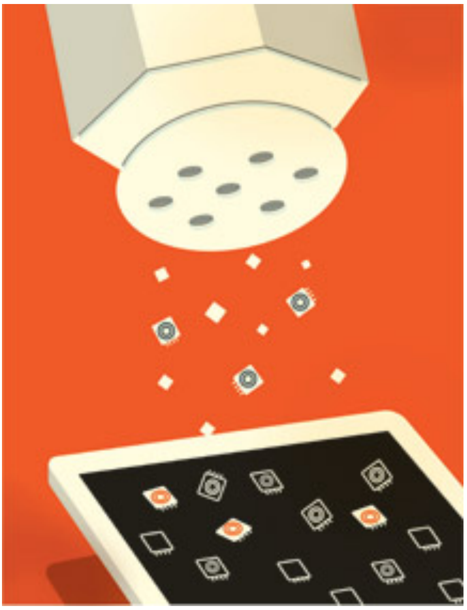


Illustration: Harry Campbell

In 1975, future Hall of Famer Roger Staubach had the football but little else in a playoff game against the Minnesota Vikings. Behind by four points at midfield with 24 seconds to go, the Dallas Cowboys quarterback closed his eyes, threw the ball as hard as he could, and said a Hail Mary. (For you soccer fans, this would be like David Beckham taking a shot on goal from midfield late in injury time.)

His prayer was answered. Staubach's receiver collided with a Viking defender just as the ball arrived but nevertheless managed to pin the football against his leg, scoring the touchdown that took the Cowboys to the Super Bowl. (Imagine Beckham's long ball beating the goalie.) Ever since that game, a desperate pass with little chance of success has been labeled a Hail Mary.

Thirty years later, the semiconductor industry threw the equivalent of a Hail Mary pass when it switched from making microprocessors run faster to putting more of them on a chip—doing so without any clear notion of how such devices would in general be programmed. The hope is that someone will be able to figure out how to do that, but at the moment, the ball is still in the air.

Why take such a gamble? In short, because there wasn't much of an alternative.

For decades, microprocessor designers used the burgeoning number of transistors that could be squeezed onto each chip to boost computational horsepower. They did this by creating microprocessors that could carry out several operations at once—for example, fetching the next instruction from memory while the current one was being executed. And chipmakers continually upped microprocessor clock rates, something the diminishing size of transistors readily allowed.

But around 2003, chipmakers found they could no longer reduce the operating voltage as sharply as they had in the past as they strived to make transistors smaller and faster. That in turn caused the amount of waste heat that had to be dissipated from each square millimeter of silicon to go up. Eventually designers hit what they call the power wall, the limit on the amount of power a microprocessor chip could reasonably dissipate. After all, a laptop that burned your lap would be a tough sell.

Designers now accept that although transistors will still get smaller and more numerous on each chip, they aren't going to operate faster than they do today. (Indeed, peak clock speeds are lower now than they were five years ago.) And if you tried to incorporate all those transistors into one giant microprocessor, you might well end up with a device that couldn't compute any faster than the chip it was replacing, which explains the shift to assembling them into multiple microprocessor cores instead. Although each core may have modest computational abilities, you'll have many of them at your disposal.

Such novel chips are called multicore microprocessors—or sometimes many-core microprocessors when a large number of cores are involved—to distinguish them from traditional single-core designs. In a sense, the core has become the new transistor, to borrow a phrase from Chris Rowen, president and chief technology officer of Tensilica, in Santa Clara, Calif. That is, from here on out, chip designers will concentrate on how to gang together lots of cores, just as the previous generation of microprocessor engineers thought about the circuitry they were creating at the level of individual transistors.

The trick will be to invent ways for programmers to write applications that exploit the increasing number of processors found on each chip without stretching the time needed to develop software or lowering its quality. Say your Hail Mary now, because this is not going to be easy.

When the president and CEO of Intel, Paul S. Otellini, announced in 2004 that his company would dedicate "all of our future product designs to multicore environments," why did he label this "a key inflection point for the industry"? The answer is clear to anyone familiar with the many now-defunct companies that bet their futures on the transition from single-core computers to systems with multiple processors working in parallel. Ardent, Convex, Encore, Floating Point Systems, Inmos, Kendall Square Research, MasPar, nCUBE, Sequent, Tandem, and Thinking Machines are just the most prominent names from a very long list of long-gone parallel hopefuls. Otellini was announcing that despite this sobering record, software applications in the future will run faster only if programmers can write parallel programs for the kinds of multicore microprocessors that Intel and other semiconductor companies have started shipping.

But why is parallel processing so challenging? An analogy helps here. Programming is in many ways like writing a news story. Potentially, 10 reporters could complete a story 10 times as fast as a single reporter could ever manage it. But they'd need to divide a task into 10 equally sized pieces; otherwise they couldn't achieve a full tenfold speedup.

Complications would arise, however, if one part of the story couldn't be written until the rest of it was done. The 10 reporters would also need to ensure that each bit of text was consistent with what came before and that the next section flowed logically from it, without repeating any material. Also, they would have to time their efforts so that they finished simultaneously. After all, you can't publish a story while you're still waiting for a piece in the middle to be completed. These same issues—load balancing, sequential dependencies, and synchronization—challenge parallel programmers.

Researchers have been trying to tackle these problems since the 1960s. Many ideas have been tried, and just about as many have failed. One early vision was that the right computer language would make parallel programming straightforward. There have been hundreds—if not thousands—of attempts at developing such languages, including such long-gone examples as APL, Id, Linda, Occam, and SISAL. Some made parallel programming easier, but none has made it as fast, efficient, and flexible as traditional sequential programming. Nor has any become as popular as the languages invented primarily for sequential programming.

Another hope was that if you just designed the hardware properly, parallel programming would become easy. Many private investors have been seduced by this idea. And many people have tried to build the El Dorado of computer architecture, but no one has yet succeeded.

A third idea, also dating back to the 1960s, is to write software that will automatically parallelize existing sequential programs. History teaches that success here is inversely proportional to the number of cores. Depending on the program, there will likely be some benefit from trying to automatically parallelize it for two, four, or even eight cores. But most experts remain skeptical that the automatic parallelization of arbitrary sequential code is going to be beneficial for 32, 64, or 128 cores, despite some recently published advances in this area.

All in all, things look pretty bleak. Nevertheless, there has been progress in some communities. In general, parallelism can work when you can afford to assemble a crack team of Ph.D.-level programmers to tackle a problem with many different tasks that depend very little on one another. One example is the database systems that banks use for managing ATM transactions and airlines use for tracking reservations. Another example is Internet searching. It's much easier to parallelize programs that deal with lots of users doing pretty much the same thing rather than a single user doing something very complicated. That's because you can readily take advantage of the inherent task-level parallelism of the problem at hand.

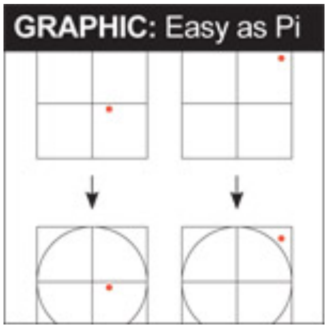
Another success story is computer graphics. Animated movies or ones with lots of computer-generated special effects exhibit task-level parallelism in that individual scenes can be computed in parallel. Clever programmers have even found parallelism in computing each image. Indeed, the high-end graphics processing units (GPUs) used to accelerate games on a PC can contain hundreds of processors, each tackling just a small piece of the job of rendering an image. Computer scientists apply the term "data-level parallelism" to such applications. They're hard enough to program, but in general they're easier than applications that don't offer this inherent parallelism.

Scientific computing provides a third success story—weather prediction and car-crash simulations being two well-known examples. These are long-running programs that have lots of data-level parallelism. The elite teams that create these programs are often combinations of Ph.D. computer scientists and people with doctorates in the sciences relevant to the application. Desktop applications rarely have that much intellectual horsepower behind them.

Given this stark landscape, you might not expect this latest foray into parallel computing to be greeted by success. But there are reasons for optimism. First off, the whole computer industry is now working on the problem. Also, the shift to parallelism is starting small and growing slowly. Programmers can cut their teeth on dual- and quad-core processors right now, rather than jumping to 128 cores in one fell swoop.

One of the biggest factors, though, is the degree of motivation. In the past, programmers could just wait for transistors to get smaller and faster, allowing microprocessors to become more powerful. So programs would run faster without any new programming effort, which was a big disincentive to anyone tempted to pioneer ways to write parallel code. The La-Z-Boy era of program performance is now officially over, so programmers who care about performance must get up off their recliners and start making their programs parallel.

Another potential reason for success is the synergy between many-core processing and software as a service, or cloud computing as it is often called. Google Search, Hotmail, and Salesforce are some here-and-now examples of such services, in which the application you need runs in a remote data center rather than on your own computer. These services are popular because they reduce the hassle to both users and providers. The user needs only a browser and needn't fool with software installation, upgrades, and patches. Software providers are happy, too, because their applications run only inside a data center where they control the environment. This has allowed their developers to improve their software much faster than can programmers writing traditional "shrink-wrap" applications, which must run on a host of different computers with many combinations of hardware and software installed.



Click on image for a larger view.

Expert programmers can take advantage of the task-level parallelism inherent in cloud computing. To service millions of users, these programmers divvy up the work to run on thousands of computers. Because their software already uses many processors, it's easy for the people mounting such operations to embrace many-core chips. Indeed, these cloud-computing providers see many-core as a welcome way to reduce costs rather than as a disruptive technology. So expect the coming proliferation of many-core processors to boost today's rapidly growing zeal for cloud computing.

Despite these reasons for hope, the odds are still against the microprocessor industry squarely completing its risky Hail Mary pass and finding some all-encompassing way to convert every piece of software to run on many parallel processors. I and other researchers at the main centers of parallel-computing research—including Georgia Tech, the University of Illinois, Rice University, Stanford, and the University of California, Berkeley—certainly don't expect that to happen. So rather than working on general programming languages or computer designs, we are instead trying to create a few important applications that can take advantage of many-core microprocessors. Although none of these groups is likely to develop the ultimate killer app, that's not the intention. Rather, we hope that the hardware and software we invent will contain some of the key innovations needed to make parallel programming straightforward. If we're successful, this work should help to usher in whatever application ultimately wins the "killer" distinction.

For example, my colleagues and I at Berkeley's parallel computing laboratory—the Par Lab—have decided to pursue just a few target applications. One is speech recognition, or perhaps I should say speech understanding. Our hope is to improve speech-recognition software dramatically so that a computer can recognize words spoken in crowded, noisy, and reverberant environments. That would surpass today's crude speech-recognition software and allow such things as real-time meeting transcription. Such software exists now, but those programs generate a frustratingly large number of mistakes.

One problem we're facing in this effort is that microprocessors with large numbers of cores are not yet being manufactured. So we have nothing to run our experimental software on. And a prototype many-core microprocessor would take years to design and millions of dollars to fabricate. We could, in principle, emulate such chips in software. But software running at the level of detail needed to evaluate a 128-core design could take days to simulate a few seconds, which means that iterations between hardware and software improvements would be excruciatingly slow.

We can, however, skirt this roadblock by using field-programmable gate arrays (FPGAs) to simulate future computers. FPGAs are integrated circuits that contain large collections of circuit components that can be wired together on the fly using a special language to describe the desired hardware configuration. And they can be rewired as many times as needed. So they offer the best of both worlds, having the flexibility of software but also the ability to run 250 times as fast as software simulators. This prospect inspired the Research Accelerator for Multiple Processors (RAMP) project, a collaboration of nearly a dozen universities and companies, to create and share a common infrastructure to accelerate research in many-core designs.

Illustration: Harry Campbell

How many RAMP configurations and specialized programming environments will be needed is still anyone's guess. In 2004, Phillip Colella of Lawrence Berkeley National Laboratory claimed that seven numerical methods would dominate scientific computing for the next decade. Starting with that claim, he, I, and a small group of other Berkeley computer scientists spent two years evaluating how well these seven techniques would work in other fields of endeavor. We ended up expanding the list to 12 general methods. Think of these as fundamental computational patterns, ones that are as different as the various motifs found in, say, Persian carpets—trees, spirals, paisley, and so forth.

Some of the 12 computational motifs are embarrassingly parallel. Take, for example, what are called Monte Carlo methods, which examine many independent random trials of some physical process to determine a more general result. You can do a lot with this approach. You could, for instance, determine the value of pi. Just compute what happens when you throw darts at a square board. If the darts hit random points on the square, what fraction of them fall within the largest circle you can draw on the board? Calculate that number for enough darts and you'll know the area of the circle. Dividing by the radius squared then gives you a value for pi.

Other motifs can be a lot harder to carry out in parallel, such as the common problem of sequencing through a series of well-defined states, where the rules for transitioning from one state to another depend on the values of various external inputs. A sequential computer calculates which state to assume next based on which state it is in and the inputs presented at that moment. Having multiple cores available doesn't do much to speed up that process, so the only opportunity to run through the sequence of states faster is to figure out ahead of time the state transitions that might be coming up. But that requires the computer to guess which state it might soon find itself in and how the inputs might change in the meantime. And when you guess wrong, it takes so much time to recover that you'll go even slower than you would have without any guessing. The hope is that you'll guess correctly most of the time, so that on average you come out ahead. Figuring out how to program such speculation about state transitions is tricky, to say the least.

In 1995, I made some public predictions of what microprocessors would be like in the year 2020. I naively expected that the information technology community would discover how to do parallel programming before chipmakers started shipping what I then called "micromultiprocessors." From the perspective of 2010, I now see three possibilities for 2020.

The first is that we drop the ball. That is, the practical number of cores per chip hits a ceiling, and the performance of microprocessors stops increasing. Such an outcome will have a broad impact on the information technology industry. Microprocessors will likely still get cheaper every year, and so will the products that contain them. But they won't pack in more computational oomph. Consider netbooks as the first step down this cost-reduction path. Such an evolution will only accelerate the shift to cloud computing, because the servers that are doing the real work will be able to take advantage of the parallelism of many-core microprocessors, even if desktops and handheld computers cannot.

Another possibility is that a select few of us will be able to catch today's risky Hail Mary pass. Perhaps only multimedia apps such as video games can exploit data-level parallelism and take advantage of the increasing number of cores. In that case, the microprocessors of 2020 may look more like the GPUs from Nvidia, Advanced Micro Devices, and Intel than the traditional microprocessors of today. That is, the GPU will be promoted from a sideshow to the main event. It's unclear whether such applications by themselves will be able to sustain the growth of the information technology industry as a whole.

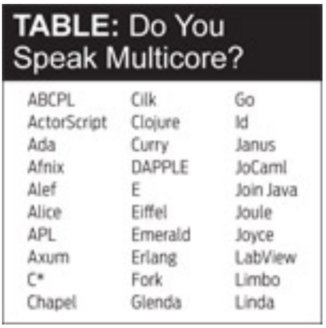


TABLE: Do You Speak Multicore?

ABCPL
ActorScript
Ada
Afnix
Alef
Alice
APL
Axum
C*
Chapel
Cilk
Clojure
Curry
DAPPLE
E
Eiffel
Emerald
Erlang
Fork
Glenda
Go
Id
Janus
JoCaml
Join Java
Joule
Joyce
LabView
Limbo
Linda

Click on image for a larger view.

The most optimistic outcome, of course, is that someone figures out how to make dependable parallel software that works efficiently as the number of cores increases. That will provide the much-needed foundation for building the microprocessor hardware of the next 30 years. Even if the routine doubling every year or two of the number of transistors per chip were to stop—the dreaded end of Moore's Law—innovative packaging might allow economical systems to be created from multiple chips, sustaining the performance gains that consumers have long enjoyed.

Although I'm rooting for this outcome—and many colleagues and I are working hard to realize it—I have to admit that this third scenario is probably not the most likely one. Just as global climate change will disadvantage some nations more than others, what happens to the microprocessor industry will probably be uneven in its effect. Some companies will succumb to the inability of microprocessors to advance in the way they have in the past. Others will benefit from the change in the new playing field of computing.

No matter how the ball bounces, it's going to be fun to watch, at least for the fans. The next decade is going to be interesting.