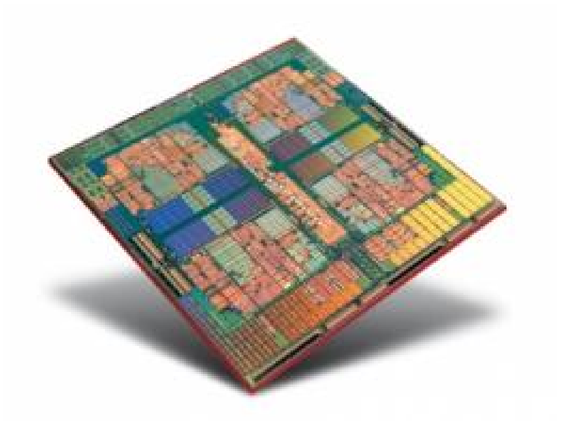
Parallel course: Researchers help ease transition to parallel programming

October 23 2009, by Larry Hardesty


An image of a Quad-Core AMD Opteron processor. Courtesy of AMD

(PhysOrg.com) -- In 1995, a good computer chip had a clock speed of about 100 megahertz. Seven years later, in 2002, a good computer chip had a clock speed of about three gigahertz -- a 30-fold increase. And now, seven years later, a good computer chip has a clock speed of... still about three gigahertz.

Four or five years ago, chip makers realized that they couldn't make chips go much faster, so they adopted a new strategy for increasing computers' power: putting multiple "cores," or processing units, on each chip. In theory, a chip with two cores working in parallel can accomplish twice as much as a chip with one core. But software developers tend to see their programs as lists of sequential instructions, and they've had trouble breaking up those instructions in ways that take advantage of parallel processing. Professor of Computer Science and Engineering Saman Amarasinghe and his colleagues at MIT's Computer Science and Artificial Intelligence Lab are giving them a hand.

In the past, computer science researchers had hoped that sequential programs could be converted into parallel programs automatically. "I spent a good part of my life trying to do that," says Amarasinghe. But Amarasinghe has now come to the conclusion that "if you want to get parallel performance, you have to start writing parallel code." At a high level, he believes, programmers will need to specify which tasks performed by their programs can run concurrently.

"Just writing anything parallel doesn't mean that it's going to run fast," Amarasinghe says. "A lot of parallel programs will actually run slower, because you parallelize the wrong place." And determining when to parallelize, and which cores to assign which tasks, is something that computers can do automatically, Amarasinghe believes.

Amarasinghe divides his group's work on multicore computing into two categories: tools to ease programmers' transition to parallel programming and tools to improve programs' performance once that transition is complete.

Predictable parallelism

The first set of tools addresses one of the central difficulties of parallel programming: if several tasks are assigned to separate cores, there's no way to be sure which will be completed first. Most of the time, that's not a problem. But occasionally, different cores have to access the same resource — updating the same stored value, for instance, or sending data to the monitor. Depending on which core reaches the resource first, the program's behavior could be quite different. Even given the exact same inputs, a multicore program may not execute in quite the same way twice in a row.

That's a nightmare for developers trying to debug their programs. To find a bug, developers run their programs over and over, with slight variations, to localize problems to ever-smaller blocks of code. But those variations don't convey any useful information unless the rest of the program behaves in the exact same way.

Amarasinghe and his grad students Marek Olszewski and Jason Ansel designed a system to make multicore programs more predictable. When a core tries to access a shared resource, it's assigned a priority based not on the time of its request but on the number of instructions it's executed. At any point during a parallel program's execution, any two cores will have executed about the same number of instructions. But if one core has had to, say, access a little-used value in a distant part of the computer's memory, it might have fallen slightly behind. In those cases, the researchers' system gives it a chance to catch up. Once it's reached the same instruction count, if it hasn't issued a higher-priority request for the shared resource, the first core's request is granted.

That waiting around means, of course, that the program as a whole will run more slowly. But in experiments, the researchers determined that, on average, a software implementation of their system increased program execution time by only about 16 percent. For developers, that's a small price to pay for reliable debugging. Moreover, if the ability to count instructions were built into the cores themselves, the system would be much more efficient. Indeed, Intel — which has demonstrated eight-core chips at trade shows — is talking with Amarasinghe about funding further research on the system.

"A lot of other people have this alternative approach," says Krste Asanovic, an associate professor at the Parallel Computing Lab at the

University of California, Berkeley. "You kind of just run it [the parallel program] however it runs and then try and record what it did so then you can go back and replay it." But with the MIT system, he says, "You don't have to worry about recording how it executed because when you execute it, it will always run the same way. So it's strictly more powerful than replay."

Maximizing multicore

Amarasinghe's lab has two projects that fit into his second category — tools that optimize the performance of parallel programs. The first deals with streaming applications, such as video broadcast over the Web. Before your computer can display an Internet video, it needs to perform a slew of decoding steps — splitting the data stream into parallel signals, performing different types of decompression and color correction, recombining the signals, performing motion compensation, equalizing the signal, and so on. Traditionally, Amarasinghe says, video software will take a chunk of incoming data, pass it through all those decoding steps, and then grab the next chunk.

That approach, says Amarasinghe, squanders the inherent parallelism of signal-processing circuits. As one chunk of data exits each decoding step, the next chunk of data could be entering it. StreamIt, a language developed by Amarasinghe's group, allows programmers to specify the order of the steps but automatically determines when to pass how much data to each step.

"If you have some kind of specification document, they're all written in block diagrams" that illustrate a signal-processing path as a series of sequential steps, Amarasinghe says. "When they write to [the programming language] C, you lose those blocks." StreamIt, by contrast, "is very close to the original thinking," Amarasinghe says.

"Other people had talked about stream programming, but StreamIt was really putting everything together in a language and compiler tool chain so people could actually write streaming programs," says Asanovic. "So I think the language design and compiler tool chain that followed from that was pretty influential."

The group's other parallel-computing project helps programs adapt on the fly to changing conditions. Often, Amarasinghe explains, a programmer has several ways of tackling a particular task. There are, for example, many common algorithms for sorting data, with names like quicksort, insertion sort, radix sort, and the like. But the algorithms perform differently under different circumstances. Quicksort is often the best choice, but not "when the data is very small," Amarasinghe says. With huge data sets, however, radix sort may work even better than quicksort.

That variability is compounded when the algorithms are being assigned to different cores. So Amarasinghe's group has designed another language that asks the developer to specify four or five different methods for performing a given computational task. When the program is running, the computer automatically identifies the most efficient method.

It might seem that that approach would lead to bloated programs. But Amarasinghe points out that when a typical program executes, it devotes much more memory to storing data than it does to storing instructions. "Something like a sort is, like, five lines of code that work on arrays with billions of elements. So making five lines of code 25 is not a big issue," he says.

It does require some added work on the programmer's part. But "there's no free lunch: you won't get nice multicore performance by doing nothing," Amarasinghe says. "Believe me, we've been trying for 50 years."

Provided by Massachusetts Institute of Technology ([news](#) : [web](#))