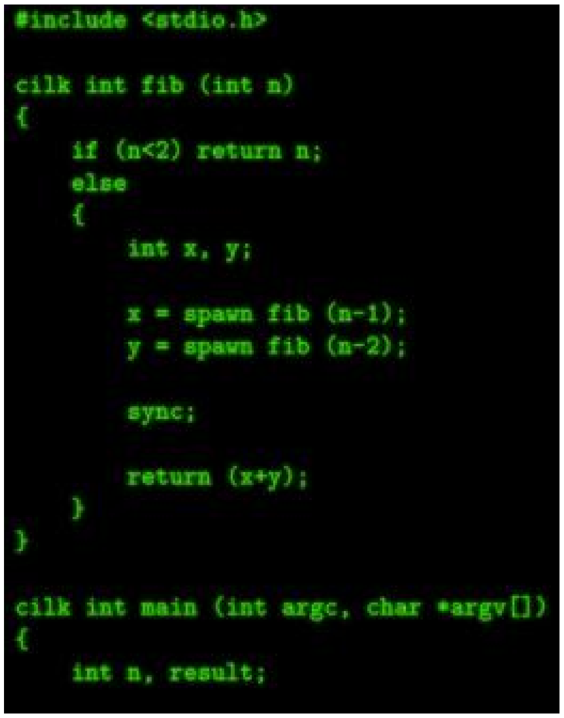
Language barrier: To take advantage of multicore chips, programmers will need fundamentally new software

March 2 2011, By Larry Hardesty



```
#include <stdio.h>

cilk int fib (int n)
{
    if (n<2) return n;
    else
    {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
    }
}

cilk int main (int argc, char *argv[])
{
    int n, result;
```

A snippet of code written in the Cilk language, showing the 'spawn' and 'sync' commands. Credit: Christine Daniloff

For decades, computer scientists tried to develop software that could automatically turn a conventional computer program -- a long sequence of instructions intended to be executed in order -- into a parallel program -- multiple sets of instructions that can be executed at the same time.

Now, most agree that that was a forlorn hope: Code that can be parallelized is too hard to recognize, and the means for parallelizing it are too diverse and context-dependent. "If you want to get parallel performance, you have to start writing parallel code," says MIT computer-science professor Saman Amarasinghe. And MIT researchers are investigating a host of techniques to make writing parallel code easier.

One of the most prominent is a software development system created by computer-science professor Charles Leiserson and his Supertech Research Group. Initially, the system used the programming language C — hence its name, Cilk. Cilk, Leiserson says, adds three commands to C: “spawn," “sync,” and a variation of the standard command “for.” If a programmer has identified a section of a program that can be executed in parallel — if, say, the same operation has to be performed on a lot of different data — he or she simply inserts the command “spawn” before it. When the program is running, the Cilk system automatically allocates the spawned computation as many cores as are free to handle it. If the results of the spawned computations need to be aggregated before the program moves on to the next instruction, the programmer simply inserts the command “sync.”

The reason Leiserson’s group could get away with such minimal alteration of C is the “runtime” system that underlies programs written in Cilk. A runtime is an extra layer of software between a program and a computer’s operating system, which allows the same program to run on much different machines; the most familiar example is probably the runtime that interprets programs written in Java. “All the smartness is underneath, in the runtime system,” Leiserson explains.

One unusual feature of Cilk’s runtime is the way it allocates tasks to different cores. Many parallel systems, Leiserson explains, use a technique called “work sharing,” in which a core with a parallel task queries the other cores on the chip to see which are free to take on some additional work. But passing messages between cores is much more time-consuming than executing computations on a given core, and it ends up eating into the gains afforded by parallel execution. The Cilk runtime instead uses a technique called "work stealing." A core that generates a host of tasks that could, in principle, be executed in parallel just queues them up in its own memory, as it would if there were no other cores on the chip. A core that finds itself without work, on the other hand, simply selects one other core at random and pulls tasks out of its queue. As long as the program has enough parallelism in it, this drastically reduces the communication overhead.

One of the advantages of Cilk, Leiserson explains, is that the programmer writes the same program whether it's going to run on a multicore computer or a single-core computer. Execution on a single-core computer is no different than execution on a computer with multiple cores, all but one of which is busy. Indeed, Cilk's advantages are evident enough that Intel now designs its compilers — the programs that convert code into instructions intelligible to computers — to work with Cilk.

Amarasinghe is attacking the problem of parallel programming on several fronts. One difficulty with parallel programs is that their behavior can be unpredictable: If, for instance, a computation is split between two cores, the program could yield a very different result depending on which core finishes its computation first. That can cause headaches for programmers, who often try to identify bugs by changing a line or two of code and seeing what happens. That approach works only if the rest of the program executes in exactly the same way. Amarasinghe and his students have developed a system in which cores report their results in an order determined by the number of instructions they've executed, not the time at which they finished their computations. If a core with a short list of instructions runs into an unexpected snag —

if, say, its request for data from main memory gets hung up — the other cores will wait for it to finish before reporting their own results, preserving the order in which the results arrive.

Another project, called StreamIt, exploits the parallelism inherent in much digital signal processing. Before a computer can display an Internet video, for instance, it needs to perform a slew of decoding steps — including several different types of decompression and color correction, motion compensation and equalization. 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. But with StreamIt, as one chunk of data is exiting a step, another chunk is entering it. The programmer just has to specify what each step does, and the system automatically divides up the data, passes it between cores, and synthesizes the results.

A programmer trying to decide how to perform a particular computation generally has a range of algorithms to choose from, and which will work best depends on the data it's handling and the hardware it's running on. Together with professor of applied mathematics Alan Edelman, Amarasinghe has developed a language called PetaBricks that allows programmers to specify different ways to perform the same computation. When a PetaBricks program launches, it performs a series of measurements to determine which types of operations will work best on that machine under what circumstances. Although PetaBricks could offer mild advantages even on single-core computers, Amarasinghe explains, it's much more useful on multicore machines. On a single-core machine, one way of performing a computation might, in rare cases, prove two or three times as efficient as another; but because of the complexities of parallel computing, the difference on a multicore machine could be a factor of 100 or more.

One of the more radical parallel-programming proposals at the

Computer Science and Artificial Intelligence Laboratory comes from in the lab of Panasonic Professor of Electrical Engineering Gerald Sussman. Traditionally, computer scientists have thought of computers as having two fundamental but distinct components: a logic circuit and a memory bank. In practice, that distinction has been complicated by evolving hardware designs, but for purposes of theoretical analysis, it's generally taken for granted.

Sussman and his former postdoc Alexey Radul, who completed his PhD at MIT in 2009 and is now at the Hamilton Institute in Maynooth, Ireland, suggest that we instead envision a computer as a fleet of simple processors and memory cells, and programming as wiring those elements together in different patterns. That conception, Radul believes, would make it easier to design software to solve problems common in artificial intelligence, such as constraint-satisfaction problems, whose solutions need to meet several sometimes-contradictory conditions at once. Sudoku puzzles are a simple example.

Radul's network is an abstraction, designed to make things easier for AI researchers: It could, in principle, be implemented on a single core. But it obviously lends itself to multicore computing. Either way, one of the central problems it poses is how to handle situations in which multiple processing units are trying to store different values in a shared memory cell. In his doctoral thesis, Radul demonstrated how to design memory cells that store information about data rather than storing the data themselves, much like a Sudoku solver who jots several possibilities in the corner of an empty square. But Radul acknowledges that designing the memory cells is just a first step in making his and Sussman's system feasible. Reinventing computing from the ground up will take more work than that.

This story is republished courtesy of MIT News *(web.mit.edu/newsoffice/), a popular site that covers news about MIT research, innovation and teaching.*

More information: Computer chips' clocks have stopped getting faster. To maintain the regular doubling of computer power that we now take for granted, chip makers have been giving chips more “cores,” or processing units. But how to distribute computations across multiple cores is a hard problem, and this five-part series of articles examines the different levels at which MIT researchers are tackling it, from hardware design up to the development of new programming languages.

Designing the hardware - www.physorg.com/news217669712.html
The next operating system - www.physorg.com/news/2011-02-t … perating-system.html
Retooling algorithms - www.physorg.com/news/2011-02-r … ling-algorithms.html
Minimizing communication between cores - www.physorg.com/news/2011-02-minimizing-cores.html

Provided by Massachusetts Institute of Technology

Citation: Language barrier: To take advantage of multicore chips, programmers will need fundamentally new software (2011, March 2) retrieved 25 April 2024 from https://phys.org/news/2011-03-language-barrier-advantage-multicore-chips.html