A reprint from

American Scientist
the magazine of Sigma Xi, The Scientific Research Society

This reprint is provided for personal and noncommercial use. For any other use, please send a request Brian Hayes by electronic mail to bhayes@amsci.org.

Computing in a Parallel Universe

Brian Hayes

THE PACE OF CHANGE in computer technology can be breathtaking—and sometimes infuriating. You bring home a new computer, and before you can get it plugged in you're hearing rumors of a faster and cheaper model. In the 30 years since the microprocessor first came on the scene, computer clock speeds have increased by a factor of a thousand (from a few megahertz to a few gigahertz) and memory capacity has grown even more (from kilobytes to gigabytes).

Through all this frenzy of upgrades and speed bumps, one aspect of computer hardware has remained stubbornly resistant to change. Until recently, that new computer you brought home surely had only one CPU, or central processing unit—the computer-within-the-computer where programs are executed and calculations are performed. Over the years there were many experiments with multiprocessors and other exotica in the world of supercomputers, but the desktops and laptops familiar to most of us continued to rely on a single-CPU architecture whose roots go back to the age of the vacuum tube and the punch card.

Now a major shift is under way. Many of the latest computers are equipped with "dual core" processor chips; they bundle two CPUs on a single slab of silicon. The two processors are meant to share the work of computation, potentially doubling the machine's power. Quad-core chips are also available; Intel has announced an eight-core product, due in 2009; Sun Microsystems has been testing a 16-core chip. A company called Tilera even offers 64 cores. It seems we are on the threshold of another sequence of doublings and redoublings, with the number of cores per chip following the same kind of exponential growth curve that earlier traced the rise in clock speed and memory capacity.

Multicore chips could bring about the biggest change in computing since the microprocessor

The next computer you bring home, a few years from now, could have hundreds or even thousands of processors. If all goes according to plan, you may notice nothing different about the new machines apart from another boost in performance. Inside, though, coordinating all those separate computational cores is going to require profound changes in the way programs are designed. Up to now, most software has been like music written for a solo performer; with the current generation of chips we're getting a little experience with duets and quartets and other small ensembles; but scoring a work for large orchestra and chorus is a different kind of challenge.

Free Lunch

Why build chips with multiple processors? Why not just keep cranking up the speed of a single CPU? If the latter option were feasible, the chipmakers would be delighted to adopt it. They are turning to multicore systems only because the path to higher gigahertz seems to be blocked, at least for now.

The causes of this impasse lie in the peculiar physical and economic laws that govern the design of integrated circuits. The most celebrated of those laws is an economic miracle: As transistors or other components are made smaller and packed more densely on the surface of a silicon chip, the cost of producing the chip remains nearly constant. Thus the cost per transistor steadily declines; it's now measured in nanodollars. This extraordinary fact is the basis of Moore's Law, formulated in 1965 by Gordon E. Moore, one of the founders of Intel. Moore observed that the number of transistors on a state-of-the-art chip doubles every year or two.

Less famous than Moore's Law but equally important are several "scaling laws" first stated in 1974 by Robert H. Dennard and several colleagues at IBM. Dennard asked: When we reduce the size of a transistor, how should we adjust the other factors that control its operation, such as voltages and currents? And what effect will the changes have on performance? He found that voltage and current should be proportional to the linear dimensions of the device, which implies that power consumption (the product of voltage and current) will be proportional to the area of the transistor. This was an encouraging discovery; it meant that even as the number of devices on a chip increased, the total power density would remain constant.

Dennard's conclusion about performance was even more cheering. In digital circuits, transistors act essentially as on-off switches, and the crucial measure of their performance is the switching delay: the time it takes to go from the conducting to the nonconducting state or vice versa. The scaling laws show that delay is proportional to size, and so as circuits shrink they can be operated at higher speed.

Taken together, these findings suggest that our universe is an especially friendly place for making electronic computers. In other realms, the laws of nature seem designed to thwart us. Thermodynamics and quantum me-

Brian Hayes is Senior Writer for American Scientist. *Additional material related to the "Computing Science" column appears in Hayes's Weblog at http://bit-player.org. Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@amsci.org*

© 2007 Brian Hayes. Reproduction with permission only.
Contact bhayes@amsci.org.

chanics tell us what we *can't* hope to do; levers amplify either force or distance but not both. Everywhere we turn, there are limits and tradeoffs, and no free lunch. But Moore's Law and the Dennard scaling rules promise circuits that gain in both speed and capability, while cost and power consumption remain constant. From this happy circumstance comes the whole bonanza of modern microelectronics.

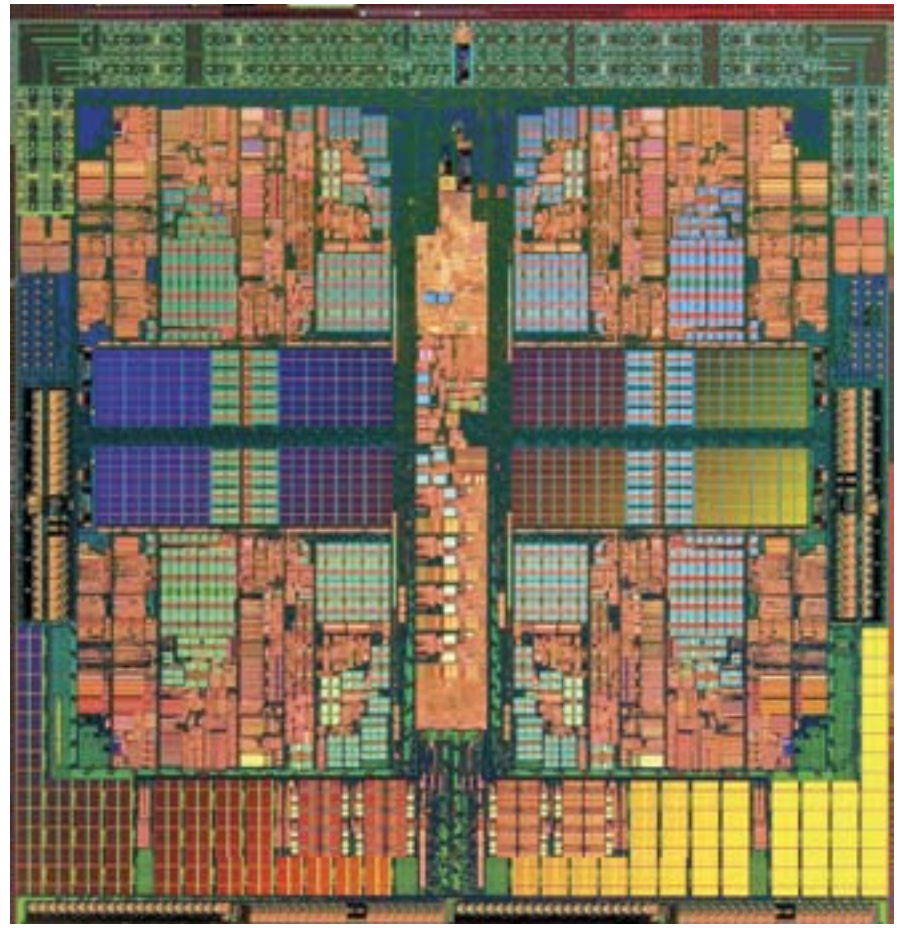
The Impasse

Free lunch is great, but there's still a bill to pay for breakfast and dinner. Throughout the past decade, chip designers have struggled with two big problems.

First, although CPUs are a thousand times faster, memory speed has increased only by a factor of ten or so. Back in the 1980s, reading a bit from main memory took a few hundred nanoseconds, which was also the time needed to execute a single instruction in a CPU. The memory and the processor cycles were well matched. Today, a processor could execute a hundred instructions in the time it takes to get data from memory.

One strategy for fixing the memory bottleneck is to transfer data in large blocks rather than single bits or bytes; this improves throughput (bits per second), but not latency (the delay before the first bit arrives). To mitigate the latency problem, computers are equipped with an elaborate hierarchy of cache memories, which surround the processor core like a series of waiting rooms and antechambers. Data and instructions that are likely to be needed immediately are held in the innermost, first-level cache, which has only a small capacity but is built for very high speed. The second-level cache, larger but a little slower, holds information that is slightly less urgent. Some systems have a third-level cache.

Reliance on cache memory puts a premium on successfully predicting which data and instructions a program is going to call for next, and there's a heavy penalty when the prediction is wrong. Moreover, processor chips have to sacrifice a large fraction of their silicon area to make room for caches and the logic circuits that control them. As the disparity between memory and CPU speed grows more extreme, a processor begins to look like a shopping mall where the stores are dwarfed by the surrounding parking lot. At some point, all the benefits of any further boost in processor speed will be eaten up by the demand for more cache.



A "quad core" microprocessor chip manufactured by Advanced Micro Devices has four separate processors that act in parallel. The cores are the four large areas of irregular geometry; most of the gridlike regions hold cache memory. The chip is part of the AMD Opteron product line and is also known by its prerelease codename Barcelona. The total silicon area of 285 square millimeters holds about 600 million transistors.

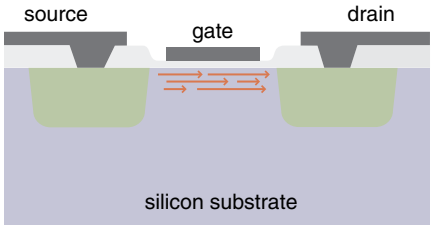
The second problem that plagues chip designers is a power crisis. Dennard's scaling laws promised that power density would remain constant even as the number of transistors and their switching speed increased. For that rule to hold, however, voltages have to be reduced in proportion to the linear dimensions of the transistor. Manufacturers have not been able to lower operating voltages that steeply.

Historically, each successive generation of processor chips has scaled the linear dimensions by a factor of 0.7, which yields an area reduction of one-half. (In other words, density doubles.) The scaling factor for voltages, however, has been 0.85 rather than 0.7, with the result that power density has been rising steadily with each new generation of chips. That's why desktop machines now come equipped with fans that could drive a wind tunnel, and laptops burn your knees.

In the future, even the 0.85 voltage reduction looks problematic. As voltage is lowered, transistors become leaky, like valves that cannot be completely shut off. The leakage current now accounts for roughly a third of total power consumption; with further reductions in voltage, leakage could become unmanageable. On the other hand, without those continuing voltage reductions, the clock rate cannot be increased.

These problems with memory latency and power density are sometimes viewed as signalling the end of Moore's Law, but that's not the apocalypse we're facing. We can still pack more transistors onto a chip and manufacture it for roughly constant cost. The semiconductor industry "roadmap" calls for increasing the number of transistors on a processor chip from a few hundred million today to more

© 2007 Brian Hayes. Reproduction with permission only.
Contact bhayes@amsci.org.



The field-effect transistor, seen here in cross section, is the building block of virtually all microelectronic circuits. A voltage applied to the gate controls the flow of current from source to drain. The transistor is fabricated by implanting ions in selected areas *(green)* and depositing layers of insulating silicon dioxide *(light gray)* and metal interconnects *(dark gray)*. The width of the gate is a crucial dimension; in recent chips it is 65 nanometers.

than 12 billion by 2020. What appears to be ending, or at least dramatically slowing, is the scaling law that allows processor speed to keep climbing. We can still have smaller circuits, but not faster ones. And hence the new Lilliputian strategy of Silicon Valley: lots of little processors working in parallel.

A Notorious Hangout

Parallel processing is hardly a new idea in computer science. Machines with multiple processors were built as early as the 1960s, when it was already widely believed that some form of "massive parallelism" was the way of the future. By the 1980s that future was at hand. David Gelernter of Yale University wrote that "parallel computing, long a notorious hangout for utopians, theorists, and backyard tinkerers, has almost arrived and is definitely for sale."

Throughout that decade and into the early 1990s novel parallel architectures became a wonderful playground for computer designers. For example, W. Daniel Hillis developed the Connection Machine, which had 2^{16} single-bit processors (and 2^{12} blinking red lights). Another notable project was the Transputer, created by the British semiconductor firm Inmos. Transputer chips were single processors designed for interconnection, with built-in communications links and facilities for managing parallel programs.

Software innovators were also drawn to the challenges of parallelism. The Occam programming language was devised for the Transputer, and languages called *Lisp and C* were written for the Connection Machine. Gelernter introduced the Linda programming system, in which multiple processors pluck tasks from a cloud called "tuple space."

What became of all these ventures? They were flattened by the steamroller of mass-market technology and economics. Special-purpose, limited-production designs are hard to justify when the same investment will buy hundreds or thousands of commodity PCs, which you can mount in racks and link together in a loose federation via Ethernet. Such clusters and "server farms" soon came to dominate large-scale computing, especially in the sciences. The vendors of supercomputers eventually gave in and began selling systems built on the same principle. All of the fastest supercomputers are now elaborations of this concept. In other words, parallelism wasn't defeated; it was co-opted.

It's also important to note that parallelism of a different kind insinuated itself into mainstream processor designs. The impressive performance of recent CPU chips comes not only from gigahertz clock rates but also from doing more during each clock cycle. The processors "pipeline" their instructions, decoding one while executing another and storing results from a third. Whenever possible, two or more instructions are executed simultaneously. Through such "instruction-level parallelism" a single CPU can have a throughput of more than one instruction per cycle, on average.

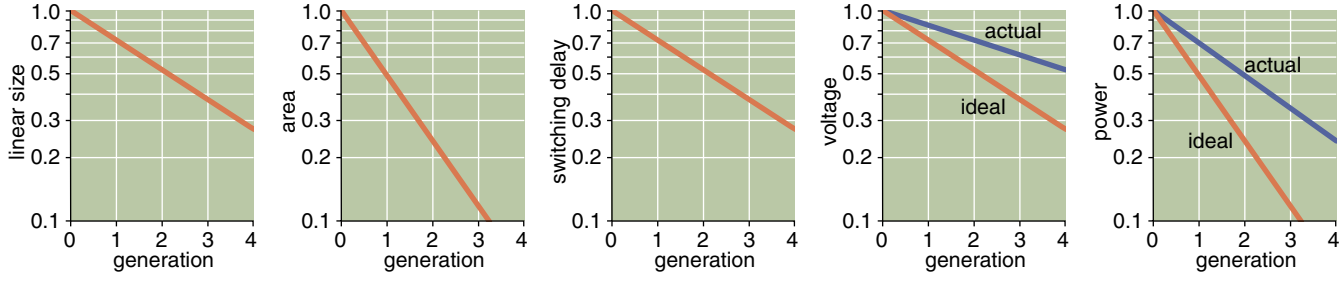
Shared Memories

It is surely no coincidence that the kinds of parallelism in widest use today are the kinds that seem to be easiest for programmers to manage. Instruction-level parallelism is all but invisible to the programmer; you create a sequential series of instructions, and it's up to the hardware to find opportunities for concurrent execution.

In writing a program to run on a cluster or server farm, you can't be oblivious to parallelism, but the architecture of the system imposes a helpful discipline. Each node of the cluster is essentially an independent computer, with its own processor and private memory. The nodes are only loosely coupled; they communicate by passing messages. This protocol limits the opportunities for interprocess mischief. The software development process is not radically different; programs are often written in a conventional language such as Fortran or C, augmented by a library of routines that handle the details of message passing.

Clusters work well for tasks that readily break apart into lots of nearly independent pieces. In weather prediction, for example, each region of the atmosphere can be assigned its own CPU. The same is true of many algorithms in graphics and image synthesis. Web servers are another candidate for this treatment, since each visitor's requests can be handled independently.

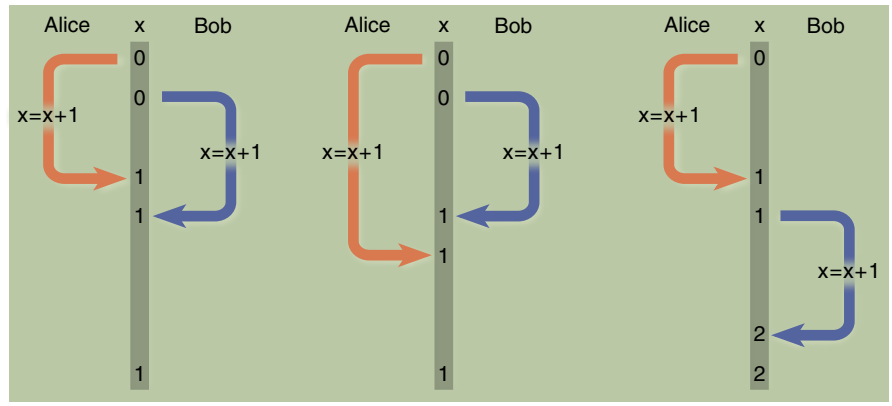
In principle, multicore computer systems could be organized in the



Scaling laws relate the physical dimensions of transistors to their electrical properties. In each successive generation of microprocessors linear dimensions (such as the gate width of a transistor) are reduced by a factor of about 0.7, which means the area of a transistor is cut in half. Switching delay (the reciprocal of processing speed) is proportional to the linear size. If operating voltage could also be lowered by a factor of 0.7, a transistor's power consumption would be proportional to its surface area, and the power density of the entire chip would remain constant. But voltages have actually been reduced only by 0.85 per generation, with the result that power and heat have become limiting factors.

© 2007 Brian Hayes. Reproduction with permission only.
Contact bhayes@amsci.org.

same way as clusters, with each CPU having its own private memory and with communication governed by a message-passing protocol. But with many CPUs on the same physical substrate, it's tempting to allow much closer collaboration. In particular, multicore hardware makes it easy to build shared-memory systems, where processors can exchange information simply by reading and writing the same location in memory. In software for a shared-memory machine, multiple computational processes all inhabit the same space, allowing more interesting and flexible patterns of interaction, not to mention subtler bugs.



Software for parallel processors is susceptible to subtle errors that cannot arise in strictly sequential programs. Here two concurrent processes both access a shared location in memory designated by the variable name *x*. Each process reads the current value of *x*, increments it by 1, and writes the new value back to the same location. The outcome depends on the way the two transactions are interleaved, and the timing of these events is not under the programmer's control. Only the rightmost case is correct.

Losing My Minds

If our universe is a peculiarly friendly place for builders of digital computers, it is not so benign for creators of programs that run concurrently on parallel hardware. Or maybe the difficulty lies in the human mind rather than in the nature of the universe.

Think of a program that reserves airline seats. Travelers access the program through a Web site that shows a diagram of the aircraft interior, with each seat marked as either vacant or occupied. When I click on seat 3A, the program first checks its database to make sure 3A is still available; if it is, I get a confirming message, and the database is updated to show that seat 3A has been taken. All's well, at least in a sequential world. But you too may be booking a seat on the same flight, and you may want 3A. If my transaction completes before your request arrives, then I'm afraid you're out of luck. On the other hand, if you are quicker with the mouse, I'm the one who will be disappointed. But what happens if the two requests are essentially simultaneous and are handled in parallel by a multiprocessing computer? Suppose the program has just assigned the seat to me but has not yet revised the database record when your request reaches the Web server. At that instant a check of the database indicates 3A is still vacant, and so we both get confirming messages. It's going to be a cozy flight!

Of course there are remedies for this problem. Programming techniques for ensuring exclusive access to resources have been known for 50 years; they are key assets in the intellectual heritage of computer science, and the airline's programmer should certainly know all about them. Many of the same issues arise even in uniprocessor systems where "time slicing" creates the illusion that multiple programs are running at the same time.

Writing correct concurrent programs is not impossible or beyond human abilities, but parallelism does seem to make extreme demands on mental discipline. The root of the difficulty is nondeterminism: Running the same set of programs on the same set of inputs can yield different results depending on the exact timing of events. This is disconcerting if your approach to programming is to try to think like a computer.

Even though the brain is a highly parallel neural network, the mind seems to be single-threaded. You may be able to walk and chew gum at the same time, but it's hard to think two thoughts at once. Consciousness is singular. In trying to understand a computer program, I often imagine myself standing at a certain place in the program text or in a flow chart. As the instructions are executed, I follow along, tracing out the program's path. I may have to jump from place to place to follow branches and loops, but at any given moment there is always one location that I can call *here*. Furthermore, I am the only actor on the stage. Nothing ever happens behind my back or out of sight. Those airline seats can't be assigned unless I assign them.

That's how it works in a sequential program. With parallel processing, the sense of single-mindedness is lost. If I try to trace the path of execution, I have to stand in many places at once. I don't know who "I" am anymore, and there are things happening all around me that I don't remember doing. "I contain multitudes," declared Walt Whitman, but for a computer programmer this is not a healthy state of mind.

Edward A. Lee, of the University of California, Berkeley, recently described the mental challenge of writing nondeterministic programs:

> A folk definition of insanity is to do the same thing over and over again and expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs.

Lee also writes:

> I conjecture that most multithreaded general-purpose applications are so full of concurrency bugs that—as multicore architectures become commonplace—these bugs will begin to show up as system failures. This scenario is bleak for computer vendors: Their next-generation machines will become widely known as the ones on which many programs crash.

Cynics, of course, will reply that computers of *every* generation have exactly that reputation.

The Slice-and-Dice Compiler

Not everyone shares Lee's bleak outlook. There may well be ways to tame the multicore monster.

One idea is to let the operating system deal with the problems of allocating tasks to processors and balancing

© 2007 Brian Hayes. Reproduction with permission only.
Contact bhayes@amsci.org.

the workload. This is the main approach taken today with time-sliced multiprocessing and, more recently, with dual-core chips. Whether it will continue to work well with hundreds of cores is unclear. In the simplest case, an operating system would adopt a one-processor-per-program rule. Thus the spreadsheet running in the background would never slow down the action in the video game on your screen. But this policy leaves processors idle if there aren't enough programs running, and it would do nothing to help any single program run faster. To make better use of the hardware, each program needs to be divided into many threads of execution.

An alternative is to put the burden on the compiler—the software that translates a program text into machine code. The dream is to start with an ordinary sequential program and have it magically sliced and diced for execution on any number of processors. Needless to say, this Vegematic compiler doesn't yet exist, although some compilers do detect certain opportunities for parallel processing.

Both of these strategies rely on the wizardry of a programming elite—those who build operating systems and compilers—allowing the rest of us to go on pretending we live in a sequential world. But if massive parallelism really is the way of the future, it can't remain hidden behind the curtain forever. Everyone who writes software will have to confront the challenge of creating programs that run correctly and efficiently on multicore systems.

Contrarians argue that parallel programming is not really much harder than sequential programming; it just requires a different mode of thinking. Both Hillis and Gelernter have taken this position, backing it up with detailed accounts drawn from their own experience. For example, Hillis and Guy L. Steele, Jr., describe their search for the best sorting algorithm on the Connection Machine. They found, to no one's surprise, that solutions from the uniprocessor world are seldom optimal when you have 65,536 processors to play with. What's more illuminating is their realization that having immediate access to every element of a large data set means you may not need to sort at all. More recently, Jeffrey Dean and Sanjay Ghemawat of Google have described a major success story for parallel programming. They and their colleagues have written hundreds of programs that run on very large clusters of computers, all using a programming model they call MapReduce.

The lesson of these examples appears to be that we shouldn't waste effort trying to adapt or convert existing software. A new computer architecture calls for a new mental model, a new metaphor. We need to rethink the problem as well as the solution. In other words, we have a historic opportunity to clean out the closet of computer science, to throw away all those dusty old sorting algorithms and the design patterns that no longer fit. We get to make a fresh start. (Be ready to buy all new software along with your new kilocore computer.)

The Helium-cooled Laptop

Although there's doubtless a multicore processor in my future (and yours), I'm not yet entirely convinced that massive parallelism is *the* direction computing will follow for decades to come. There could be further detours and deviations. There could be a U-turn.

The multicore design averts a power catastrophe, but it won't necessarily break through the memory bottleneck. All of those cores crammed onto a single silicon chip have to compete for the same narrow channel to reach off-chip main memory. As the number of cores increases, contention for memory bandwidth may well be the factor that limits overall system performance.

In the present situation we have an abundance of transistors available but no clear idea of the best way to make use of them. Lots of little processors is one solution, but there are alternatives. One idea is to combine a single high-performance CPU and several gigabytes of main memory on the same sliver of silicon. This system-on-a-chip is an enticing possibility; it would have benefits in price, power and performance. But there are also impediments. For one thing, the steps in fabricating a CPU are different from those that create the highest-density memories, so it's not easy to put both kinds of devices on one chip. There are also institutional barriers: Semiconductor manufacturers tend to have expertise in microprocessors or in memories but not in both.

Finally, we haven't necessarily seen the last of the wicked-fast uniprocessor. The power and memory constraints that have lately driven chipmakers to multicore designs are not fundamental physical limits; they are merely hurdles that engineers have not yet learned to leap. New materials or new fabrication techniques could upset all our assumptions.

A year ago, IBM and Georgia Tech tested an experimental silicon-germanium chip at a clock rate of 500 gigahertz—more than a hundred times the speed of processors now on the market. Reaching that clock rate required cooling the device to 4 Kelvins, which might seem to rule it out as a practical technology. But which is harder: Writing reliable and efficient parallel software, or building a liquid-helium cooler for a laptop computer? I'm not sure I know the answer.

Bibliography

Agarwal, Anant, and Markus Levy. 2007. Thousand-core chips: The kill rule for multicore. In *Proceedings of the 44th Annual Conference on Design Automation DAC '07*, pp. 750–753. New York: ACM Press.

Asanovic, Krste, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick. 2006. The landscape of parallel computing research: A view from Berkeley. University of California, Berkeley, Electrical Engineering and Computer Sciences Technical Report UCB/EECS-2006-183. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

Brock, David C., ed. 2006. *Understanding Moore's Law: Four Decades of Innovation*. Philadelphia: Chemical Heritage Press.

Carriero, Nicholas, and David Gelernter. 1990. *How to Write Parallel Programs: A First Course*. Cambridge, Mass.: The MIT Press.

Dean, Jeffrey, and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, pp. 137–150. http://labs.google.com/papers/mapreduce.html

Dennard, Robert, Fritz Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous and Andre LeBlanc. 1974. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid State Circuits* SC-9(5):256–268.

Hillis, W. Daniel, and Guy L. Steele, Jr. 1986. Data parallel algorithms. *Communications of the ACM* 29:1170–1183.

Intel Corporation. 2007. Special issue on terascale computing. *Intel Technical Journal* 11(3). http://www.intel.com/technology/itj/

International Technology Roadmap for Semiconductors. 2005. http://www.itrs.net/Links/2005ITRS/Home2005.htm

Lee, Edward A. 2006. The problem with threads. *IEEE Computer* 39(5):33–42.

Sutter, Herb, and James Larus. 2005. Software and the concurrency revolution. *ACM Queue* 3(7):54–62.

© 2007 Brian Hayes. Reproduction with permission only.
Contact bhayes@amsci.org.