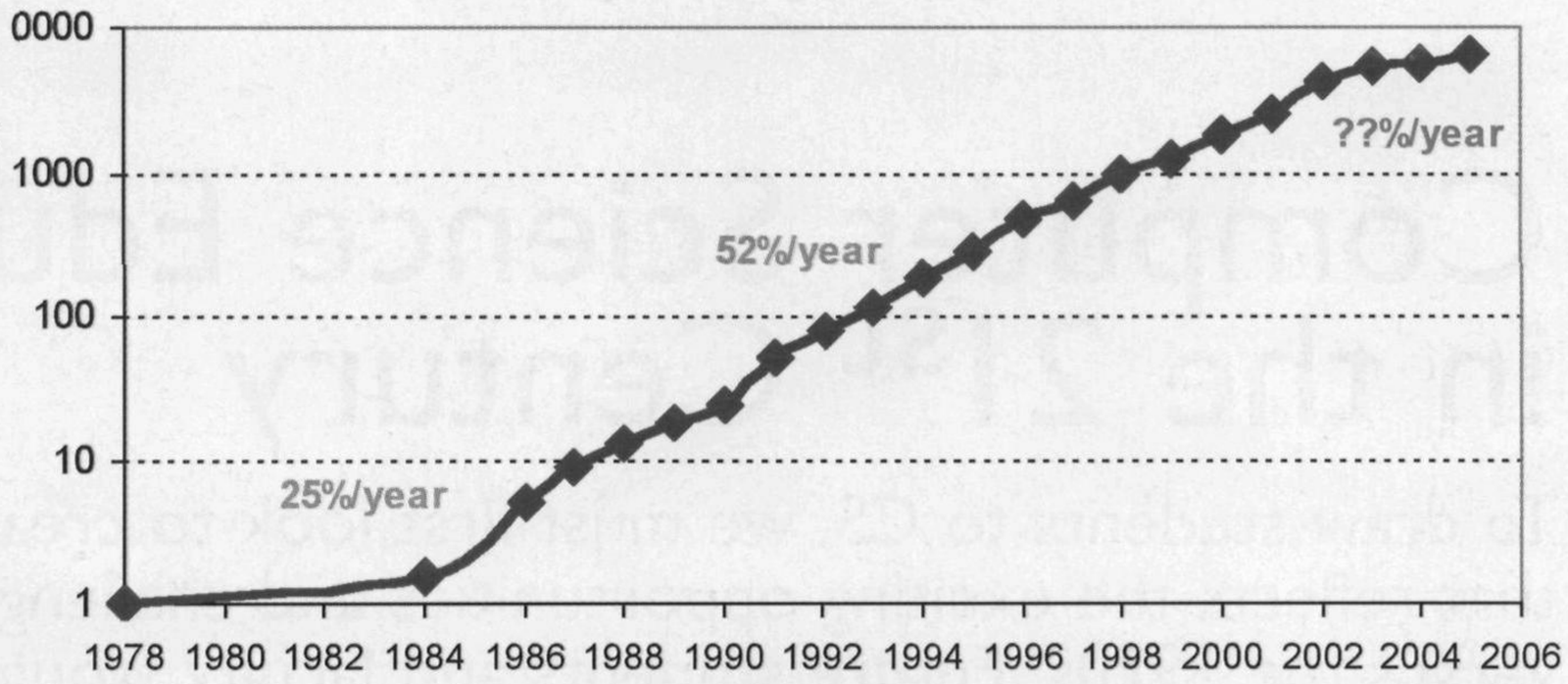
are either already shipping multiprocessors on a chip as their mainline product or will in near future. Apple just announced the next generation of Macintoshes, and all will include two processors—even laptops!

The table on the next page shows the number of processors ("cores") on a chip and number of hardware-supported threads per processor on current and upcoming microprocessors.

Hardware companies will market the number of processors per chip rather than clock rate from now on, and the number of processors is predicted to double every two years. If the prediction holds, by the time an entering freshman class graduates, standard desktop PCs will include at least eight processors.



Growth in processor performance since 1978 relative to the VAX 11/ 780 as measured by the SPECint benchmarks (from SPECint89 to SPECint2000). Prior to the mid-1980s, processor performance growth was largely technology driven, and averaged about 25% per year. The increase in growth from 1986 to 2002 to about 52% was due to more advanced architectural and organizational ideas exploiting Moore's Law. It's less than 20% per year since 2002 [3].

In our past, some have prematurely claimed that parallel programming was the only option for the immediate future. The steep part of the graph belies that claim. The difference today is that no hardware company is planning to sell much faster sequential processors. If you can't buy them, parallel procrastination will be penalized instead of rewarded.

Given this sea change, how much of the curriculum—and what fraction of the CS faculty—is oblivious to concurrency? How many algorithms, data structures, languages, compilers, debuggers, operating systems, books, and lectures must change to match the transformation in underlying technology if they are to remain relevant in the 21st century?

Educating faculty so they can train students properly is not a trivial bootstrapping problem. At research universities, if we can ensure that all future research assumes concurrent computing, the skills gained there hopefully can trickle down quickly to the classroom.

Course I Would Love to Take #1: *Join the Open Source Movement.*

Most schools teach "write programs from blank sheet of paper" programming, of which there is very little real-world bearing. A different approach is to leverage high-quality examples of the open source movement. For example, a database class might be based on PostgreSQL and an operating system class might use BSD Unix. (Well-documented and well-crafted code trumps popular for pedagogic reasons.) The high-level idea is taking advantage of a technology that has been created for other reasons that can make the classroom a much more exciting and realistic place.

If this works, it would be inspiring for students working on real production software. Even companies that don't use open source software may benefit from students who can do more than just write programs from scratch. In addition, it could be a differentiator on a college campus. Do civil engineering students get to contribute to the construction of a real bridge in the classroom? Do history courses allow students to help write a book? The recruiting pitch is to join CS and learn in part by contributing immediately to the real world.

To help learn a large system, writing documentation for portions of open source code could be an assignment. Documentation is important yet rare in the classroom and the open source movement. It's likely that open source developers would welcome good documentation, given its absence. Students would be inspired that their coursework could be