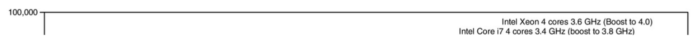
1.8 The sea change: The switch from uniprocessors to multiprocessors

> 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.
> *Brian Hayes, Computing in a Parallel Universe, 2007.*

The power limit has forced a dramatic change in the design of microprocessors. The figure below shows the improvement in response time of programs for desktop microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to a factor of 1.2 per year.

Figure 1.8.1: Growth in processor performance since the mid-1980s (COD Figure 1.17).

This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see COD Section 1.10 (Fallacies and pitfalls)). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. The higher annual performance improvement of 52% since the mid-1980s meant performance was about a factor of seven larger in 2002 than it would have been had it stayed at 25%. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 22% per year.



Rather than continuing to decrease the response time of one program running on the single processor, as of 2006 all desktop and server companies are shipping microprocessors with multiple processors per chip, where the benefit is often more on throughput than on response time. To reduce confusion between the words processor and microprocessor, companies refer to processors as "cores," and such microprocessors are generically called multicore microprocessors. Hence, a "quadcore" microprocessor is a chip that contains four processors or four cores.

PARTICIPATION ACTIVITY 1.8.1: Slowing in uniprocessor performance has led to a switch to multiprocessor systems.

Start ☐ 2x speed

PARTICIPATION ACTIVITY 1.8.2: Changes in processor design.

1) From the mid-1980s to early-2000s, processor performance improved each year at an average of 52%.
   - ○ True
   - ○ False

2) Growth in processor performance slowed in 2002.
   - ○

True

False

3) Power was a factor in the slowing of processor performance growth.

True

False

4) Manufacturers continue to design single processor systems and increase processor performance through new technology-driven improvements.

True

False

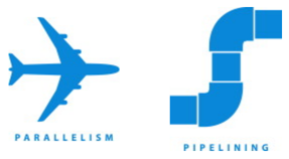
In the past, programmers could rely on innovations in hardware, architecture, and compilers to double performance of their programs every 18 months without having to change a line of code. Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of multiple processors. Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve the performance of their code as the number of cores increases.

To reinforce how the software and hardware systems work together, we use a special section, Hardware/Software Interface, throughout the book, with the first one appearing below. These elements summarize important insights at this critical interface.

Hardware/Software Interface

Parallelism has always been crucial to performance in computing, but it was often hidden. COD Chapter 4 (The Processor) will explain *pipelining*, an elegant technique that runs programs faster by overlapping the execution of instructions. This optimization is one example of instruction-level parallelism, where the parallel nature of the hardware is abstracted away so the programmer and compiler can think of the hardware as executing instructions sequentially.

Forcing programmers to be aware of the parallel hardware and to rewrite their programs to be parallel had been the "third rail" of computer architecture, for companies in the past that depended on such a change in behavior failed (see COD Section 6.15 (Historical perspective and further reading)). From this historical perspective, it's startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.



Why has it been so hard for programmers to write explicitly parallel programs? The first reason is that parallel programming is by definition performance programming, which increases the difficulty of programming. Not only does the program need to be correct, solve an important problem, and provide a useful interface to the people or other programs that invoke it; the program must also be fast. Otherwise, if you don't need performance, just write a sequential program.

The second reason is that fast for parallel hardware means that the programmer must divide an application so that each processor has roughly the same amount to do at the same time, and that the overhead of scheduling and coordination doesn't fritter away the potential performance benefits of parallelism.

As an analogy, suppose the task was to write a newspaper story. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must schedule the sub-tasks. If anything went wrong and just one reporter took longer than the seven others did, then the benefits of having eight writers would be diminished. Thus, we must balance the load evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. You would also fall short if one part of the story, such as the conclusion, couldn't be written until all the other parts were completed. Thus, care must be taken to reduce communication and synchronization overhead. For both this analogy and parallel programming, the challenges include scheduling, load balancing, time for synchronization, and overhead for communication between the parties. As you might guess, the challenge is stiffer with more reporters for a newspaper story and more processors for parallel programming.

PARTICIPATION ACTIVITY 1.8.3: Programs for multicore systems.

1) As computing systems move to multicore microprocessors, programmers _____ to obtain performance benefits.

don't need to change any code

need to rewrite their programs

2) Parallel programming seeks to improve program _____.

pipelining

performance

correctness

3) How should programmers write code to maximize the benefits of parallel programming?

Run all program tasks on a single processor

○ Run programs in a round-robin fashion to ensure even wear of processors

○ Divide a program into sub-tasks so all processors run about the same amount of time

4) Parallel programming becomes more difficult as the number of processor cores increases.

○ True

○ False

To reflect this sea change in the industry, the next five chapters in this edition of the book each has a section on the implications of the parallel revolution to that chapter:

- *COD Chapter 2 (Instructions: Language of the Computer) Section 2.11: Parallelism and Instructions: Synchronization.* Usually independent parallel tasks need to coordinate at times, such as to say when they have completed their work. This chapter explains the instructions used by multicore processors to synchronize tasks.
- *COD Chapter 3 (Arithmetic for Computers), Section 3.6: Parallelism and Computer Arithmetic: Subword Parallelism.* Perhaps the simplest form of parallelism to build involves computing on elements in parallel, such as when multiplying two vectors. Subword parallelism takes advantage of the resources supplied by *Moore's Law* to provide wider arithmetic units that can operate on many operands simultaneously.
- *COD Chapter 4 (The Processor), Section 4.10: Parallelism via Instructions.* Given the difficulty of explicitly parallel programming, tremendous effort was invested in the 1990s in having the hardware and the compiler uncover implicit parallelism, initially via *pipelining*. This chapter describes some of these aggressive techniques, including fetching and executing multiple instructions concurrently and guessing on the outcomes of decisions, and executing instructions speculatively using *prediction*.
- *COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), Section 5.10: Parallelism and Memory Hierarchies: Cache Coherence.* One way to lower the cost of communication is to have all processors use the same address space, so that any processor can read or write any data. Given that all processors today use caches to keep a temporary copy of the data in faster memory near the processor, it's easy to imagine that parallel programming would be even more difficult if the caches associated with each processor had inconsistent values of the shared data. This chapter describes the mechanisms that keep the data in all caches consistent.
- *COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), Section 5.11: Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks.* This section describes how using many disks in conjunction can offer much higher throughput, which was the original inspiration of ***Redundant Arrays of Inexpensive Disks*** (***RAID***). The real popularity of RAID proved to be the much greater dependability offered by including a modest number of redundant disks. The section explains the differences in performance, cost, and dependability between the various RAID levels.

In addition to these sections, there is a full chapter on parallel processing. COD Chapter 6 (Parallel Processor from Client to Cloud) goes into more detail on the challenges of parallel programming; presents the two contrasting approaches to communication of shared addressing and explicit message passing; describes a restricted model of parallelism that is easier to program; discusses the difficulty of benchmarking parallel processors; introduces a new simple performance model for multicore microprocessors; and, finally, describes and evaluates four examples of multicore microprocessors using this model.

As mentioned above, COD Chapters 3 (Arithmetic for Computers) to 6 (Parallel Processor from Client to Cloud) use matrix vector multiply as a running example to show how each type of parallelism can significantly increase performance.

COD Appendix B (Graphics and Computing GPUs) describes an increasingly popular hardware component that is included with desktop computers. The ***graphics processing unit*** (***GPU***) is a hardware component that accelerates graphics. GPUs are becoming programming platforms in their own right. As you might expect, given these times, GPUs rely on *parallelism*.

COD Appendix B (Graphics and Computing GPUs) describes the NVIDIA GPU and highlights parts of its parallel programming environment.

Provide feedback on this section