Computer Science 507
Software Engineering
The College of Saint Rose
Spring 2015

Topic Notes: Parallel and Distributed Computing

Why Parallel Computing?

Parallel computing sounds simple enough – when one computer isn't powerful enough to solve your problem, use more than one.

Before we start to think about how to use parallelism on a computer, let's think about a parallel approach to solving a "real-world" problem.

- Taking a census of Albany's Pine Hills neighborhood.

  One person doing this would visit each house, count the people, and ask whatever questions are supposed to be asked. This person would keep running counts. At the end, this person has gathered everything.

  If there are two people, they can work concurrently. Each visits some houses, and they need to "report in" along the way or at the end to combine their information. But how to split up the work?

  - Each person could do what the individual was originally doing, but would check to make sure each house along the way had not yet been counted.
  - Each person could start at a special office at Saint Rose, get an address that has not yet been visited, go visit it, then go back to the office to report the result and get another address to visit. Someone at the office keeps track of the cumulative totals. This is nice because neither person will be left without work to do until the whole thing is done. This is the *master-slave* method of breaking up the work.
  - The neighborhood could be split up beforehand. Each could get a randomly selected collection of addresses to visit. Maybe one person takes all houses with even street numbers and the other all houses with odd street numbers. Or perhaps one person would take everything north of Route 20 and the other everything south of Route 20. The choice of how to divide up the neighrborhood may have a big effect on the total cost. There could be excessive travel if one person walks right past a house that has not yet been visited. Also, one person could finish completely while the other still has a lot of work to do. This is a *domain decomposition* approach.

- Grading a stack of exams. Suppose each has several questions. Again, assume two graders to start.

  - Each person could take half of the stack. Simple enough. But we still have the potential of one person finishing before the other.
  - Each person could take a paper from the "ungraded" stack, grade it, then put it into the "graded" stack.
  - Perhaps it makes more sense to have each person grade half of the *questions* instead of half of the exams, maybe because it would be unfair to have the same question graded by different people. Here, we could use variations on the approaches above. Each takes half the stack, grades his own questions, then they swap stacks.
  - Or we form a *pipeline*, where each exam goes from one grader to the next to the finished pile. Some time is needed to start up the pipeline and drain it out, especially if we add more graders. These models could be applied to the census example, if different census takers each went to every house to ask different questions.
  - Suppose we also add in a "grade totaler and recorder" person. Does that make any of the approaches better or worse?

- Adding two $1,000,000 \times 1,000,000$ matrices.

  - Each matrix entry in the sum can be computed independently, so we can break this up any way we like. Could use the master-slave approach, though a domain decomposition would probably make more sense. Depending on how many processes we have, we might break it down by individual entries, or maybe by rows or columns.

In each of these cases, we have taken what we might normally think of as a *sequential* process, and taken advantage of the availability of *concurrent processing* to make use of multiple workers (processing units).

Parallelism adds complexity, so why bother?

- we want to solve the same problem but in a shorter time than possible on one processor – goal: speedup
- we want to solve larger problems than can currently be solved at all on a single processor – goal: scale-up
- some algorithms are more naturally expressed or organized concurrently
- and now: that's where performance gains come from in modern processors!

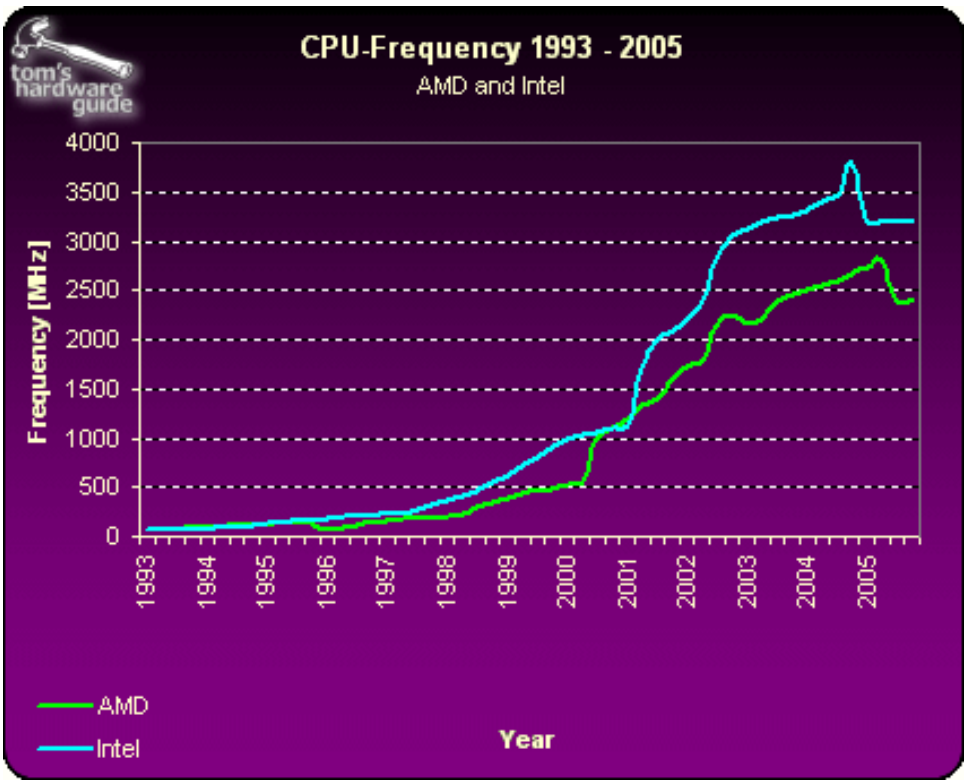


Figure used with permission from article *The Mother of All CPU Charts 2005/2006*, Bert Töpelt, Daniel Schuhmann, Frank Völkel, Tom's Hardware Guide, Nov. 2005, `http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/`

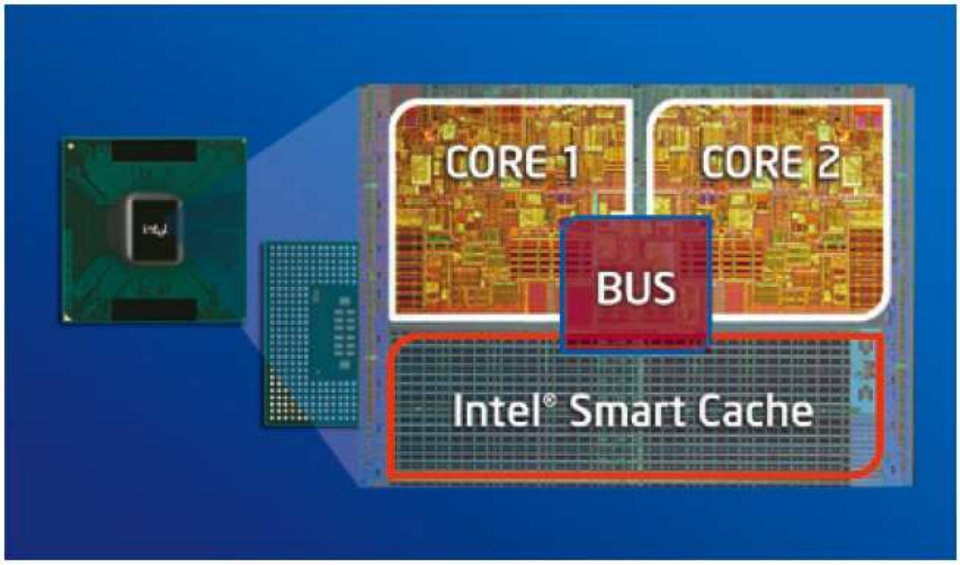


Image from Intel Core Duo Processor product brief.

Some Basics

Sequential Program: sequence of actions that produce a result (statements + variables), called a process, task, or thread (of control). The state of the program is determined by the code, data, and a *single* program counter.

Concurrent Program: two or more processes that work together. Big difference: *multiple* program counters.

To cooperate, the processes need *communication* and *synchronization*, which can be achieved through *shared variables*, or *message passing*

Hardware to run concurrent processes

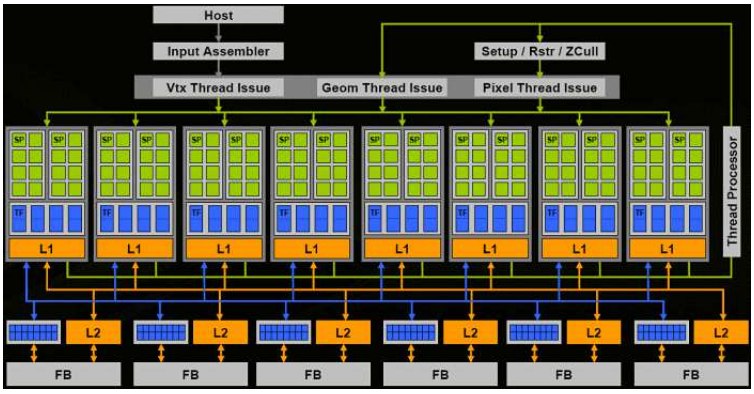
- single processor – logical concurrency (see Operating System course)
- multiprocessor – shared memory
- multicomputer – separate memories
- network – slower communication

Computers may be classified as:

- SISD: single instruction, single data – one processor doing one thing at a time to one piece of data at a time.
- SIMD: single instruction, multiple data – multiple processors all doing the same thing at the same time, but operating on different data. Also known as: vector computers. Program operates in "lock step" on each processor.
- MIMD: multiple instruction, multiple data – multiple processors each doing their own thing.
- SPMD: single program, multiple data – not really a classification of the computer, but of a model used to program a MIMD computer. Multiple processors run the same program, but do not operate in lock step. Also known as the "interacting peers" model. This is the model we will use most in this class.

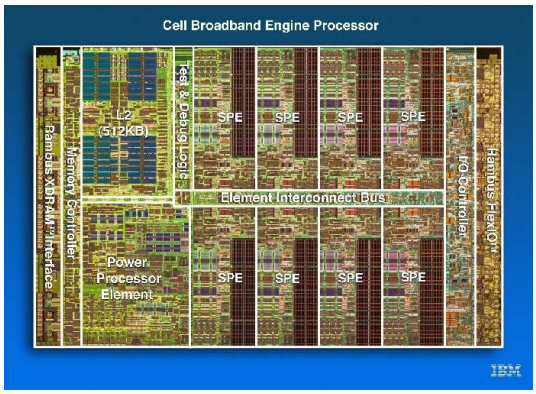
Some examples:

- SISD: Pre-"multi-core" desktops and laptops.
- SIMD: graphics cards that apply a single operation to an array of data points at the same time.



NVIDIA image

- MIMD: desktops and laptops with multiprocessors or multi-core chips – each processor can be executing any instruction and operating on any data.
- MIMD: Cell architecture (Sony PS3) – one general purpose processor and several special-purpose cores.



`http://www.research.ibm.com/cell/cell_chip.html`

- MIMD: ASCI Red, Sandia National Labs: 4600+ nodes, each with 2 Intel Pentium II Xeon processors, first TeraOp machine in 1997.
- MIMD: ASCI White, LLNL: 512 nodes, each with 16 Power3 Nighthawk-2 processors, 12 TeraOps total, was number 1 until 2002.
- Hybrid: Earth Simulator, Yokohama Institute for Earth Sciences, Japan: 640-node NEC system, each node with 8 vector processors, total of 5,120 CPUs, peak performance of 40 TeraOps
- Hybrid: IBM Blue Gene systems – dense clusters of Cell processors.

See `http://www.top500.org/`.

Moral: from the desktop to the world's largest supercomputers, it's a world of parallel processing out there!

How to Achieve Parallelism

- We need to determine where concurrency is possible, then break up the work accordingly
- This is easiest if a compiler can do this for you – take your sequential program and extract the concurrency automatically. This is sometimes possible, especially with fixed-size array computations.
- If the compiler can't do it, it is possible to give "hints" to the compiler to tell it what is safe to parallelize.
- But often, the parallelization must be done explicitly: the programmer has to create the threads or processes, assign work to them, and manage necessary communication.

Finding Concurrency

We find opportunities for parallelism by looking for parts of the sequential program that can be run in any order.

Before we look at the matrix-matrix multiply, we step back and look at a simpler example:

```
1: a = 10;
2: b = a + 5;
3: c = a - 3;
4: b = 7;
5: a = 3;
6: b = c - a;
7: print a, b, c;
```

Which statements can be run in a different order (or concurrently) but still produce the same answers at the end?

- 1 has to happen before 2 and 3, since they depend on `a` having a value.
- 2 and 3 can happen in either order.
- 4 has to happen after 2, but it can happen before 3.
- 5 has to happen after 2 and 3, but can happen before 4.
- 6 has to happen after 4 (so 4 doesn't clobber its value) and after 5 (because it depends on its value)
- 7 has to happen last.

This idea can be formalized, but it is beyond the scope of our discussion. Bottom line, some things can be done in different orders or concurrently, and other things need to happen in a specific (relative) order.

The degree to which we can allow the work of our programs to run concurrently determines how much parallelism we can hope to achieve.

Approaches to Parallelism

Automatic parallelism is great, when it's possible. If we buy a fancy parallelizing compiler, we get our parallelism for free (at least once we bought the compiler)! It does have limitations, though:

- some potential parallelization opportunities cannot be detected automatically
- bigger complication – this executable cannot run on distributed-memory systems

Parallel programs can be categorized by how the cooperating processes communicate with each other:

- **Shared Memory** – some variables are accessible from multiple processes. Reading and writing these values allow the processes to communicate.
- **Message Passing** – communication requires explicit messages to be sent from one process to the other when they need to communicate.

These are functionally equivalent given appropriate operating system support. For example, one can write message-passing software using shared memory constructs, and one can simulate a shared memory by replacing accesses to non-local memory with a series of messages that access or modify the remote memory.

Continue with talk slides...