

Parallel Computing: MultiProcessing, MultiThreading, and MultiCore

Mike Bailey

Oregon State University



"If you were plowing a field, which would you rather use –
two strong oxen or 1024 chickens?"

-- Seymore Cray



Oregon State University
Computer Graphics

The Problem

We demand increasing performance for desktop applications. How can we get that? There are four approaches that we're going to discuss here:

1. We can increase the clock speed (the "La-Z-Boy approach").
2. We can combine several separate computer systems, all working together (multiprocessing).
3. We can develop a single chip which contains multiple CPUs on it (multicore).
4. We can look at where the CPU is spending time waiting, and give it something else to do while it's waiting (multithreading).



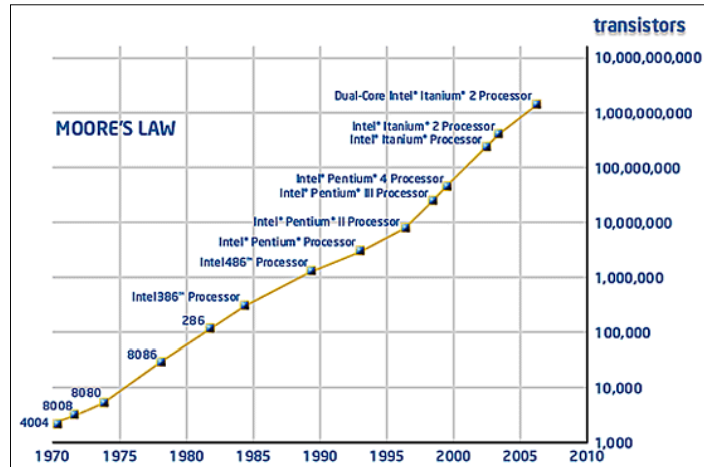
Oregon State University
Computer Graphics

mjb – November 13, 2009



1. Increasing Clock Speed -- Moore's Law

“Transistor density doubles every 1.5 years.”



Note that oftentimes people (incorrectly) equivalence this to:
“Clock speed doubles every 1.5 years.”



Oregon State University
Computer Graphics

Source: <http://www.intel.com/technology/mooreslaw/index.htm>

mjb – November 13, 2009

Moore's Law

- From 1986 to 2002, processor performance increased 52%/year (CAGR)
- Fabrication process sizes (“gate pitch”) have recently fallen from 65 nm to 45 nm to 32 nm to 22 nm
- For example, the AMD Phenom-2 uses a 45 nm process and has 758M transistors. The Phenom-1 used a 65 nm process and had 463M transistors, using a slightly larger die size.
- Next should be 16 nm !



Oregon State University
Computer Graphics

Clock Speed and Power Consumption

| | | |
|------|-----------|--------------------|
| 1981 | IBM PC | 5 MHz |
| 1995 | Pentium | 100 MHz |
| 2002 | Pentium 4 | 3000 MHz (3 GHz) |
| 2007 | | 3800 MHz (3.8 GHz) |
| 2009 | | 4000 MHz (4.0 GHz) |

Clock speed has hit a wall, largely because of power consumption.

$$\text{PowerConsumption} \propto \text{ClockSpeed} \times \text{Voltage}^2$$

But, the supply voltage you need to provide is proportional to the clock speed, so really:

$$\text{PowerConsumption} \propto \text{ClockSpeed}^3$$

Yikes!



Oregon State University
Computer Graphics

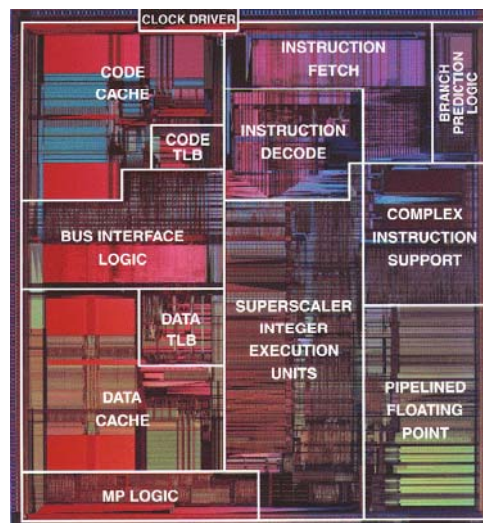
mjb - November 13, 2009

Clock Speed and Clock Skew

There is another reason that clock speeds have gotten difficult to dramatically increase:

A CPU chip is divided into logical sections, all of which must be able to interact with each other. The clock pulse comes in on a single pin of the chip housing, which is then routed all over the chip.

As clock speeds have increased, it is harder to keep the clock signal synchronized all over the chip. This is known as **clock skew**.

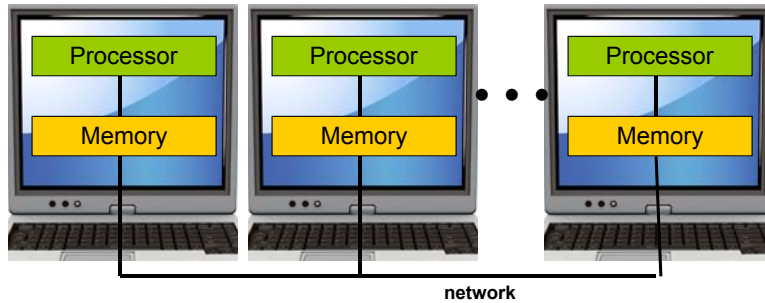


Oregon State University
Computer Graphics

mjb - November 13, 2009

2. Multiprocessing with Distributed Memory

Instead of one computer in a system, put in more than one. Distribute the overall work to the posse of computers. This is called *Multiprocessing*.



Because each processor and its memory are distributed across a network, this is oftentimes referred to as *Distributed Computing* or *Cluster Computing*.

This is interesting for many things, such as large semi-autonomous calculations. But desktop games and simulation are not like that. They need more help locally.

3. Eliminating Wasted Time: Thread-Level Parallelism

You are trying to watch two football games on TV at the same time. The commercials come at unpredictable times and are exceedingly long and annoying. It's a waste of your time to watch them.

What strategy do you follow to maximize your football viewing and minimize the commercials?

You could either:

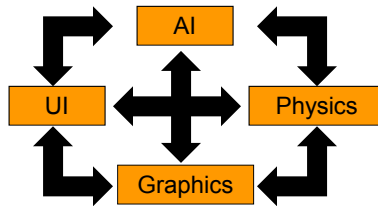
1. Just watch one, and put up with the commercials as wasted time.
2. Flip back and forth at fixed time intervals (e.g., every minute)
3. Watch one until it goes to a commercial, then flip to the other.

3. Thread Level Parallelism (TLP)

The code gets divided into multiple operations. When one operation blocks, or its time slice is over, switch to another operation.

Sometimes a program can be naturally subdivided into *independent* operations. Web serving and transaction processing are good examples.

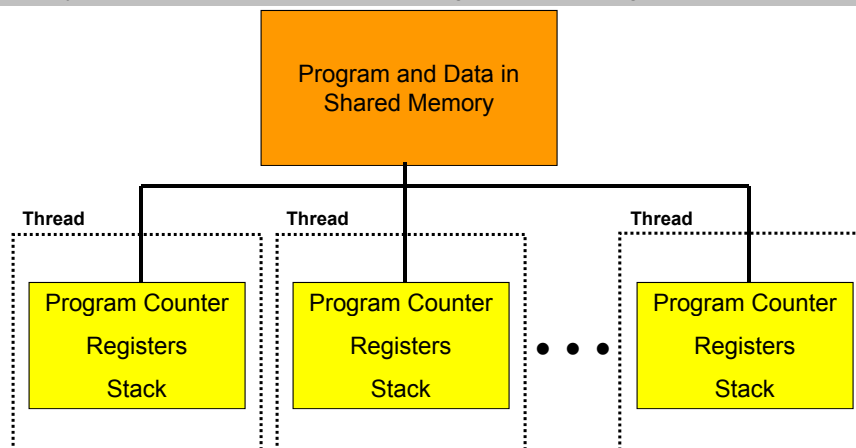
Or, sometimes programs can be naturally divided into *cooperating* operations. In a game, for example, there might be the User Interface, the AI, Physics, and the Graphics all working independently, but cooperating with each other.



How could you write this as a single program and give each operation its own time slice?

What Exactly is a Thread?

Threads are separate processes, all executing a common program and sharing memory. Each thread has its own state (program counter, registers, and stack).



What Exactly is a Thread?

A “thread” is an independent path through the program code. Each thread has its own program counter, registers, and stack. But, since each thread is executing some part of the same program, each thread has access to the same memory. Each thread is scheduled and swapped just like any other process.

Threads can share time on a single processor. You don't have to have multiple processors (although you can – multicore is our next topic).

This is useful, for example, in a web browser when you want several things to happen autonomously:

- User interface
- Communication with an external web server
- Web page display
- Animation



Oregon State University
Computer Graphics

mjb – November 13, 2009

When is it Good to use Multithreading?

- Where specific operations can become blocked, waiting for something else to happen
- Where specific operations can be CPU-intensive
- Where specific operations must respond to asynchronous I/O, including the user interface (UI)
- Where specific operations have higher or lower priority than other operations
- Where performance can be gained by overlapping I/O
- To manage independent behaviors in interactive simulations
- When you want to accelerate a single program on multicore CPU chips

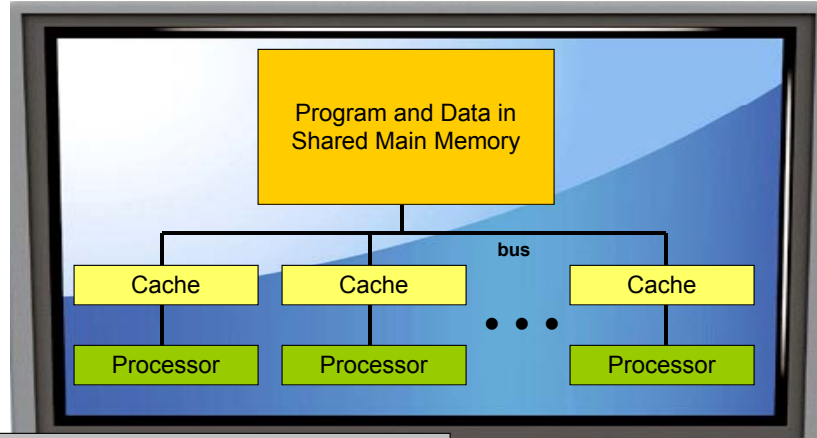


Oregon State University
Computer Graphics

mjb – November 13, 2009

4. Clever Architectures: Multiprocessing and Shared Memory

Multiprocessing, can also be accomplished on a single system through shared memory. This is called, not surprisingly, *Shared Memory Multiprocessing*.



Because each processor has equal access to the whole memory, this is oftentimes referred to as *Symmetric Multiprocessing (SMP)* or *Unified Memory Access (UMA)*

This originally was done with multiple CPU chips. But, adding all those CPU chips became expensive!



er 13, 2009

However, Multiprocessors are Not a Free Lunch: Amdahl's Law

If you put in N processors, you should get N times speedup, right?

Wrong!

There are always some amount of operations that are serial and cannot be parallelized no matter what you do. These include reading data, setting up calculations, control logic, etc.

If you think of the operations that a program needs to do as divided between a fraction that is parallelizable and a fraction that isn't (i.e., is stuck at being sequential), then **Amdahl's Law** says:

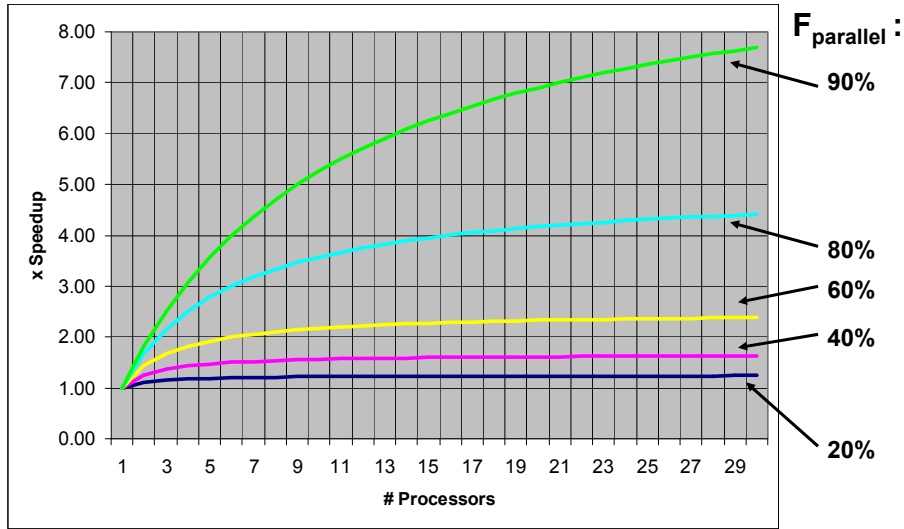
$$xSpeedup = \frac{1}{\frac{F_{parallel}}{\# processors} + F_{sequential}}$$



Oregon State University
Computer Graphics

mjb - November 13, 2009

Amdahl's Law as a Function of Number of Processors and $F_{parallel}$



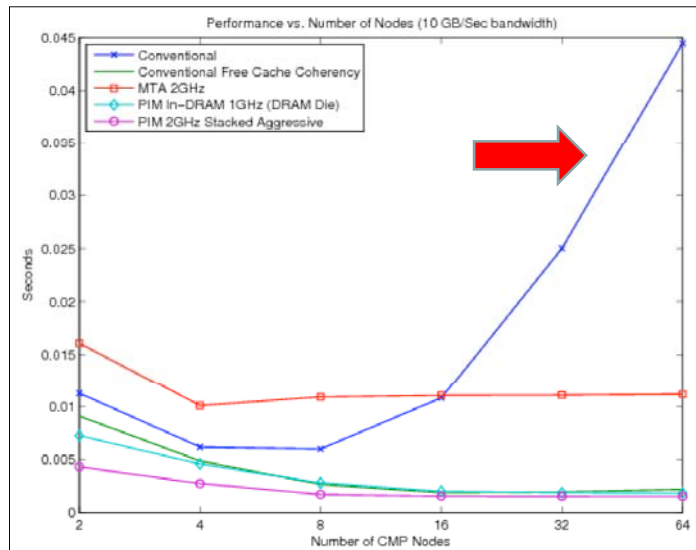
Amdahl's Law

Note that these fractions put an upper bound on how much benefit you will get from adding more processors:

$$\max Speedup = \lim_{\# processors \rightarrow \infty} xSpeedup = \frac{1}{F_{sequential}} = \frac{1}{1 - F_{parallel}}$$

| Fparallel | maxSpeedup |
|-----------|------------|
| 0.00 | 1.00 |
| 0.10 | 1.11 |
| 0.20 | 1.25 |
| 0.30 | 1.43 |
| 0.40 | 1.67 |
| 0.50 | 2.00 |
| 0.60 | 2.50 |
| 0.70 | 3.33 |
| 0.80 | 5.00 |
| 0.90 | 10.00 |
| 0.95 | 20.00 |
| 0.99 | 100.00 |

It's actually worse – if you increase the number of processors without limit, at some point they start to get in each other's way!



Oregon State University
Computer Graphics

Source: Sandia National Labs

mjb – November 13, 2009

MultiCore -- Multiprocessing on a Single Chip

So, to summarize:

1. Moore's Law of transistor density is still going strong, but the Moore's Law of clock speed has hit a wall.
2. Multiple CPU chips are an option, but are too expensive for desktop applications. Now what do we do?

Keep packing more and more transistors on a single chip, but don't increase the clock speed. Instead, increase computational throughput by using those transistors to pack multiple processors on the same chip.

Originally this was called *single chip multiprocessing*, but now it is usually referred to as **multicore**.



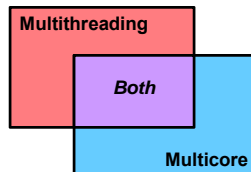
Oregon State University
Computer Graphics

mjb – November 13, 2009

MultiCore

Multicore, even without multithreading too, is still a good thing. It can be used, for example, to allow multiple programs on a desktop system to always be executing.

But, the big hope for multicore is that it is a way to speed up a *single program*. For this, we need a **combination of both multicore and multithreading**.



Multicore is a very hot topic these days. The chip vendors are implementing all new chips this way. We, as programmers, can no longer take the La-Z-Boy approach to getting program speedups.

We need to be prepared to convert our programs to run on **MultiThreaded Shared Memory Multicore** architectures.



Oregon State University
Computer Graphics

mjb - November 13, 2009

On a Multicore system, what would cause a Thread to Block (and thus allow another thread to run)?

- End of its time slice
- Encountering a Mutual Exclusion lock (mutex) that another thread owns – more on this soon
- Waiting for the user to do something in the user interface
- Disk or network I/O
- Waiting for the graphics FIFOs to drain
- Memory accesses

Even Memory Accesses Take Time to Complete

| Type of Access → | L1 Cache | L2 Cache | Main Memory |
|------------------|-----------|----------|-------------|
| Size → | 2 * 64 KB | 1 MB | "∞" |
| Cycles → | 1 | 8 | 80 |

Source: AMD

For this reason, in my opinion, cache management is a big unacknowledged headache in multicore programming!



Oregon State University
Computer Graphics

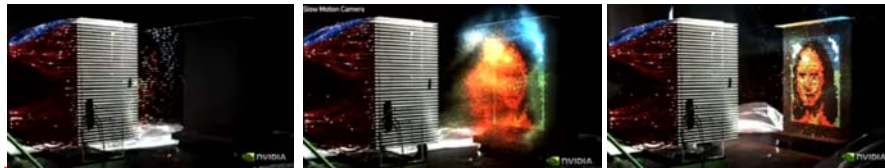
mjb - November 13, 2009

A Special Case of Multiprocessing, Multicore, and Multithreading: Data-Level Parallelism (DLP)

Data-Level Parallelism refers to achieving more performance by dividing up the data and performing identical operations on each piece. It's different from breaking the code into multiple operations – the code stays intact – but each instance of the code operates on a different piece of data.

Sometimes this is called SPMD – Single Program Multiple Data.

Graphics GPUs work this way. For example, the Fragment (pixel-producing) Processor applies identical code to the production of each individual pixel, but is fed with different data that describes what it is supposed to do at each pixel.

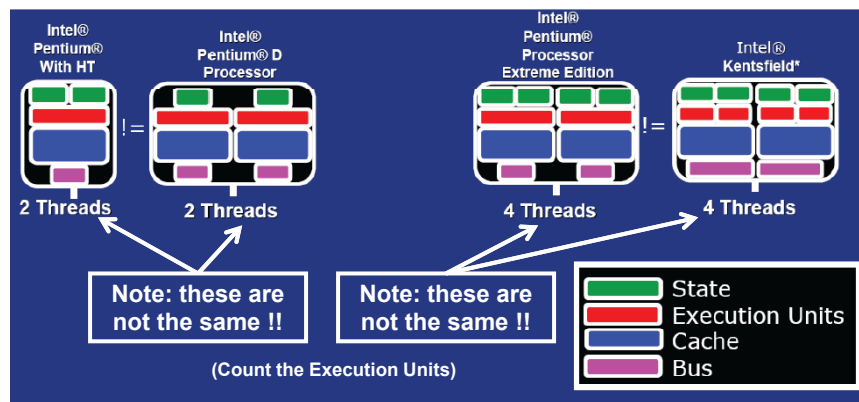


Oregon State University
Computer Graphics

<http://www.youtube.com/watch?v=ZrJeYFxpUyQ>

mjb – November 13, 2009

A Comparison of Intel Processors



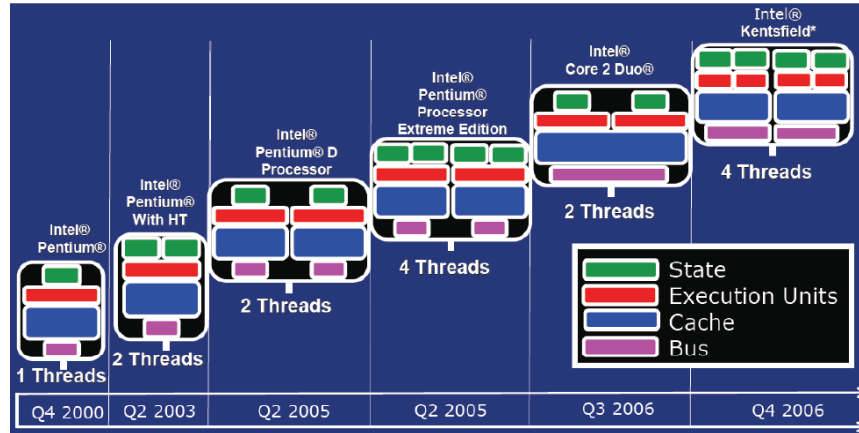
Source: Intel



Oregon State University
Computer Graphics

mjb – November 13, 2009

A Comparison of Intel Processors



Source: Intel



Oregon State University
Computer Graphics

mjb - November 13, 2009

There are many ways to perform Multicore/Multithreading Programming – the Bane of them all is *Debugging*

Deadlock and Livelock Faults

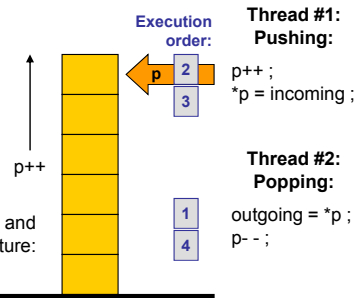
- *Deadlock*: Two threads are each waiting for the other to do something
- *Livelock*: like Deadlock, but both threads are changing state in sync with each other, possibly to avoid deadlock, and thus are still deadlocked

A good example is the dreaded hallway encounter

Race Condition Fault

- One thread modifies a variable while the other thread is in the midst of using it

A good example is maintaining and using the pointer in a stack data structure:



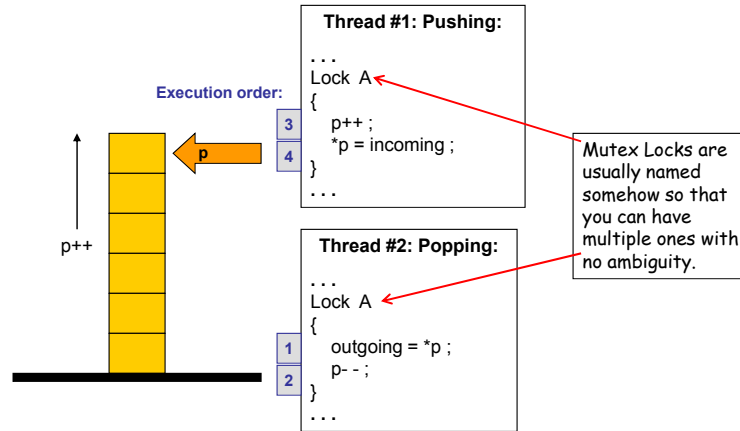
Worse yet, these problems are not always deterministic!



Oregon State University
Computer Graphics

mjb - November 13, 2009

Race Conditions can often be fixed through the use of Mutual Exclusion Locks (Mutexes)



Note that, while solving a race condition, we can also create a new deadlock condition if the thread that owns the lock is waiting for the other thread to do something

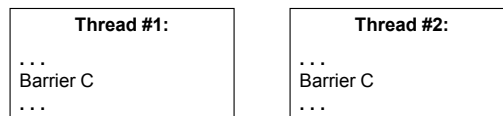


Barriers

A *barrier* is a way to let all threads get to the same point before moving on together.

For example, it is a common parallel numeric technique to solve a large matrix $[A]\{x\} = \{b\}$ by letting each thread solve a smaller sub-matrix, share its results across its boundaries, re-solve the sub-matrix, re-share, ...

But, each thread might not reach the "sharing point" at the same time. You need all the threads to wait at that point until everyone else gets there, then proceed with the sharing and re-solving.



Multithreaded Programming: How to Start

Design your program so it will work even if only one thread is available, i.e., do not require concurrency.

Then, run it with only one thread (there are calls to do this). This makes it lots easier to debug because you don't have deadlock, livelock, and race conditions.

Then, run it with two threads, four threads, etc.

Be sure to use thread-safe utility functions!

- Many standard C functions are not thread-safe
- An example is *strtok*, which keeps internal state
- Sometimes there are alternative thread-safe versions (see the documentation)
- Unfortunately, many of the Standard Template Library functions are not thread-safe



Oregon State University
Computer Graphics

mjb - November 13, 2009