Parallel Computing: MultiThreading and MultiCore

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

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

-- Seymore Cray

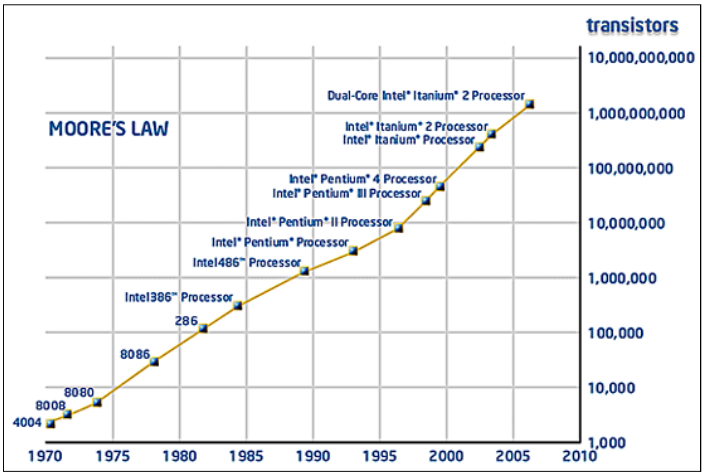
The Problem

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).

1. Increasing Clock Speed -- Moore's Law

"Transistor density doubles every 1.5 years."

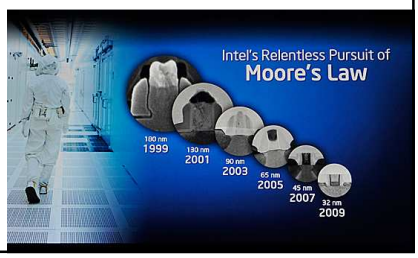


Note that oftentimes people (*incorrectly*) equivalence this to: "Clock speed doubles every 1.5 years."

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

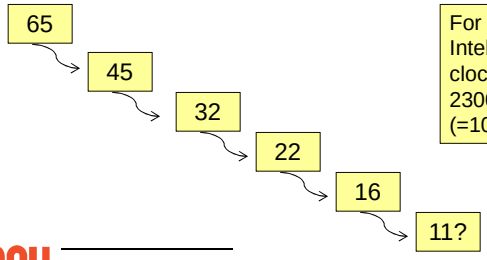
Moore's Law

- From 1986 to 2002, processor performance increased an average of 52%/year
- Fabrication process sizes ("gate pitch") have fallen from 65 nm to 45 nm to 32 nm to 22 nm
- Next will be 16 nm !



Moore's Law

- From 1986 to 2002, processor performance increased an average of 52%/year which means that it didn't quite double every 1.5 years, but it did go up by 1.87, which is close.
- Fabrication process sizes ("gate pitch") have fallen from 65 nm to 45 nm to 32 nm.
- For example, the Intel Core i7-965 used a 45 nm process and had 731M transistors. The Intel Core i7-980X uses a 32 nm process and has 1,170M transistors.
- The Intel Ivy Bridge processor uses 22 nm
- Next should be 16 nm, then 11nm !



For the sake of an entertaining comparison, the Intel 4004 processor, introduced in 1971, had a clock speed of 108 KHz (=.000108 GHz), had 2300 transistors, and used a gate pitch of 10 µm (=10,000 nm) !

Clock Speed and Power Consumption

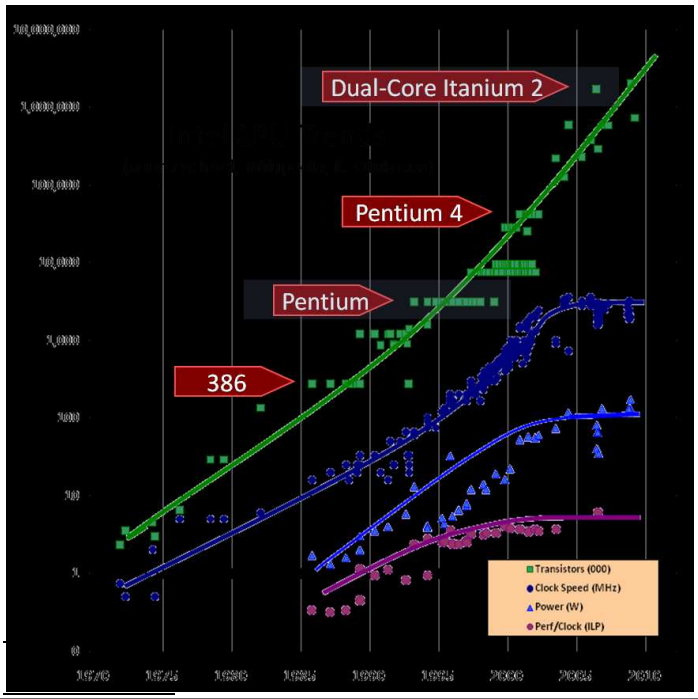
1981	IBM PC	5 MHz
1995	Pentium	100 MHz
2002	Pentium 4	3000 MHz (3 GHz)
2007		3800 MHz (3.8 GHz)
2009		4000 MHz (4.0 GHz)

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

$$PowerConsumption \propto ClockSpeed \times Voltage^2$$

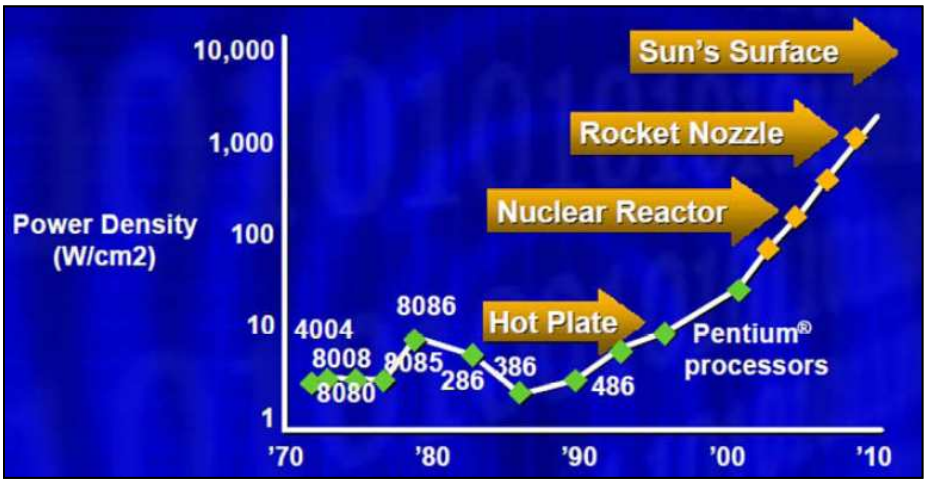
is-proportional-to

Yikes!



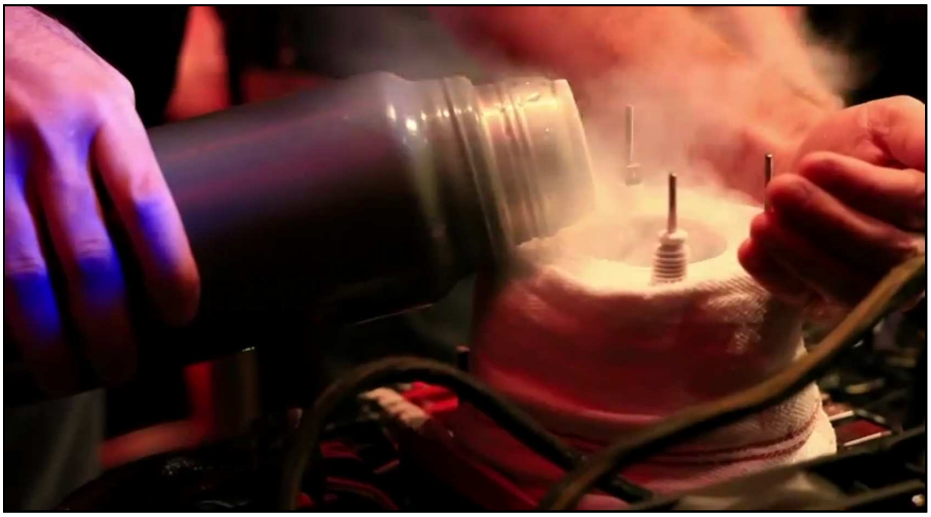
Source: Intel

What Kind of Power Density Would it Have Taken to Keep up with Clock Speed Trends?



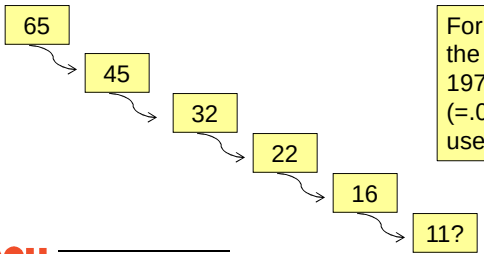
Source: Patrick Gelsinger, ***2004 Intel Developer's Forum***.

Recently, AMD set the world record for clock speed (8.429 GHz) using a Liquid Nitrogen-cooled CPU



Why Multicore? Moore's Law

- From 1986 to 2002, processor performance increased an average of 52%/year which means that it didn't quite double every 1.5 years, but it did go up by 1.87, which is close.
- Fabrication process sizes ("gate pitch") have fallen from 65 nm to 45 nm to 32 nm.
- For example, the Intel Core i7-965 used a 45 nm process and had 731M transistors. The Intel Core i7-980X uses a 32 nm process and has 1,170M transistors.
- Intel says 22 nm CPUs ????????.
- Next should be 16 nm, then 11nm !



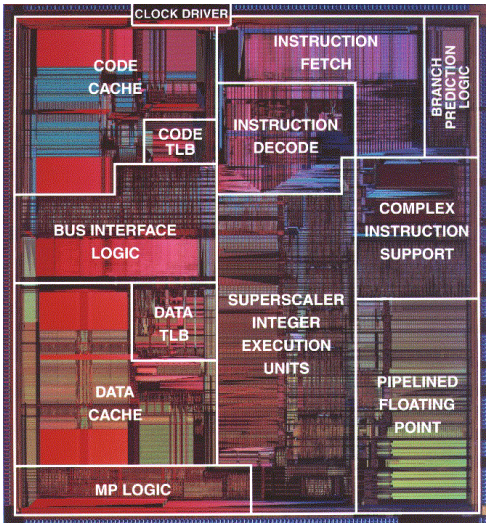
For the sake of an entertaining comparison, the Intel 4004 processor, introduced in 1971, had a clock speed of 108 KHz (=.000108 GHz), had 2300 transistors, and used a gate pitch of 10 μm (=10,000 nm) !

Clock Speed and Clock Skew

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

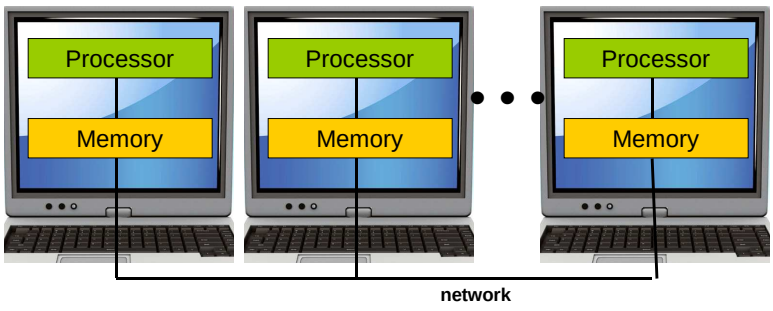
A CPU chip is divided into logical sections, all of which must be able to interact with each other. The clock pulse comes in on a single pin of the chip housing, which is then routed all over the chip.

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



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:

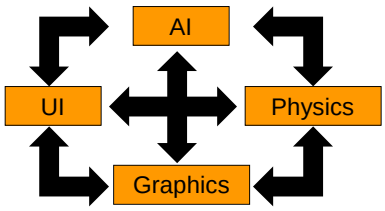
1. Just watch one, and put up with the commercials as wasted time.
2. Flip back and forth at fixed time intervals (e.g., every minute)
3. Watch one until it goes to a commercial, then flip to the other.

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.



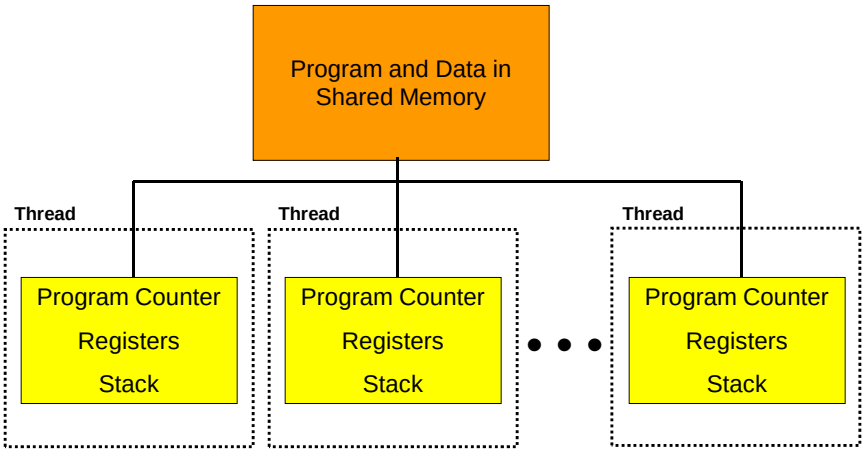
How could you write this as a single program and give each operation its own time slice?

OSU Oregon State University Computer Graphics

mjb – November 13, 2012

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

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

Threads at their Very Best: Multiprocessing + Shared Memory → Multicore

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. All these processors share the same memory.

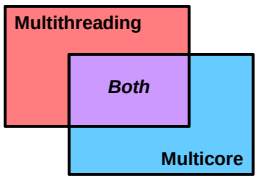
Originally this was called *single chip multiprocessing*, but now it is referred to as ***multicore***. Because each processor has equal access to the whole memory, this is oftentimes referred to as *Symmetric Multiprocessing* (SMP) or *Unified Memory Access* (UMA).

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.

Multithreading, even without multicore too, is still a good thing. Threads can make it easier to logically have many things going on in your program at a time, and can absorb the dead-time of other threads.

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.

However, Multicore is not a Free Lunch: Amdahl's Law

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

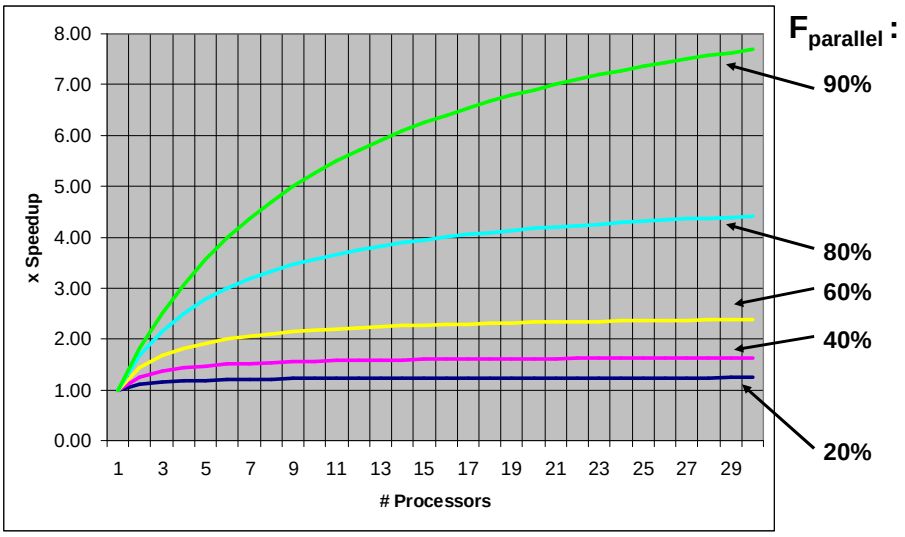
Wrong!

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:

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

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



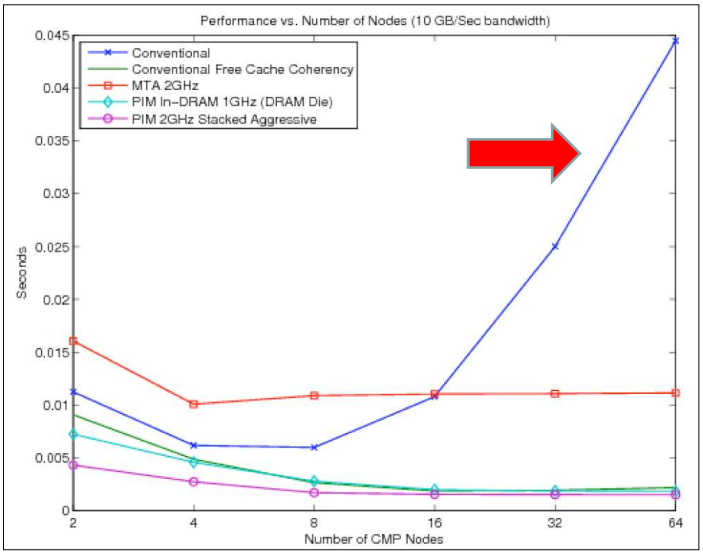
Amdahl's Law

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

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

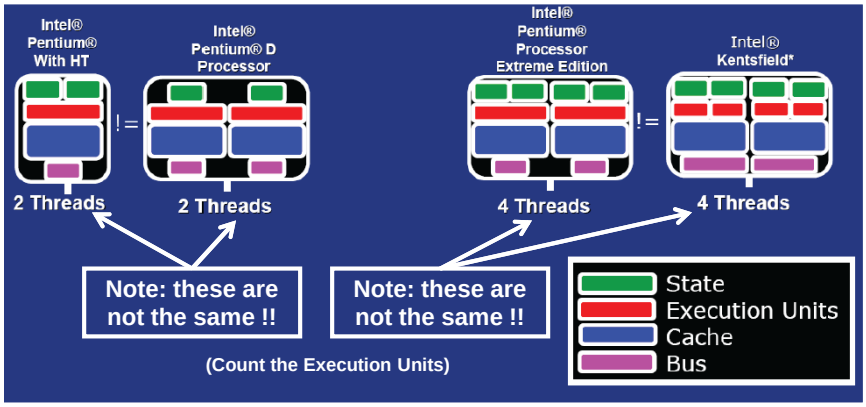
Fparallel	maxSpeedup
0.00	1.00
0.10	1.11
0.20	1.25
0.30	1.43
0.40	1.67
0.50	2.00
0.60	2.50
0.70	3.33
0.80	5.00
0.90	10.00
0.95	20.00
0.99	100.00

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!



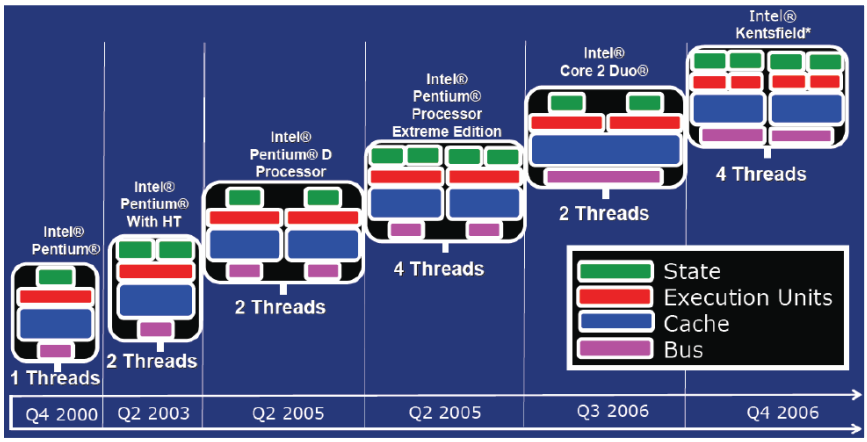
Source: Sandia National Labs

A Comparison of Intel Processors



Source: Intel

A Comparison of Intel Processors



Source: Intel

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

Deadlock and Livelock Faults

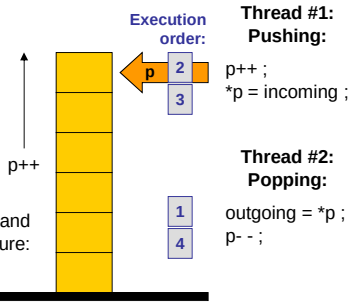
- *Deadlock*: Two threads are each waiting for the other to do something
- *Livelock*: like Deadlock, but both threads are changing state in sync with each other, possibly to avoid deadlock, and thus are still deadlocked

A good example is the dreaded hallway encounter

Race Condition Fault

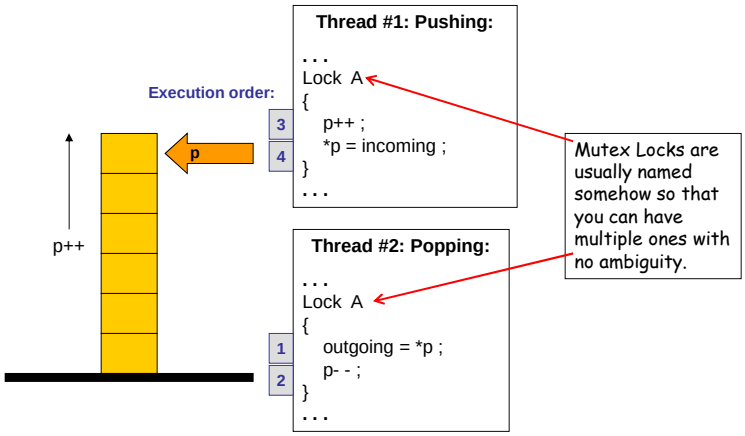
- One thread modifies a variable while the other thread is in the midst of using it

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



Worse yet, these problems are not always deterministic!

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.

Thread #1:
. . .
Barrier C
. . .

Thread #2:
. . .
Barrier C
. . .

OpenMP Multithreaded Programming

• OpenMP is a multi-vendor standard

The OpenMP paradigm is to issue C/C++ "pragmas" to tell the compiler how to build the threads into the executable

```
#pragma omp directive [clause]
```

All threads share a single global heap (malloc, new)

Each thread has its own stack (procedure arguments, local variables)

OpenMP probably gives you the biggest multithread benefit per amount of work put in to using it

Creating OpenMP threads in Loops

```
#include <omp.h>
int i;

#pragma omp parallel for private(i)
for( i = 0; i < num; i++ )
{
        . . .
}
```

This tells the compiler to parallelize the for loop into multiple threads, and to give each thread its own personal copy of the variable *i.* But, you don't have to do this for variables defined in the loop body:

```
#pragma omp parallel for
for( int i = 0; i < num; i++ )
{
        . . .
}
```

Creating Sections of OpenMP Threads

```
#pragma omp parallel sections
{
        #pragma omp section
        {
                . . .
        }
        #pragma omp section
        {
                . . .
        }
}
```

This tells the compiler to place each section of code into its own thread

If each section contains a procedure call, then this is a good way to approximate the pthreads paradigm

Number of OpenMP threads

Two ways to specify how many OpenMP threads you want to have available:

1. Set the OMP_NUM_THREADS environment variable
2. Call omp_set_num_threads(num);

Asking how many cores this program has access to:

```
num = omp_get_num_procs( );
```

Setting the number of threads to the exact number of cores available:

```
num = omp_set_num_threads(   omp_get_num_procs( )   );
```

Asking how many OpenMP threads this program is using:

```
num = omp_get_num_threads( );
```

Asking which thread this one is:

```
me = omp_get_thread_num( );
```

Data-Leval Parallelism (DPL) in OpenMP

These last two calls are especially important if you want to do Data-Level Parallelism (DLP) !

```
total = omp_get_num_threads( );
#pragma omp parallel private(me)
    me = omp_get_thread_num( );
    DoWork( me, total );
#pragma omp end parallel
```


Enabling OpenMP in Visual Studio

1. To enable OpenMP in VS, go to the Project menu → Project Properties
2. Change the setting Configuration Properties → C/C++ → Language → OpenMP Support to "Yes (/openmp)"

More on Creating OpenMP threads in Loops

Normally, variables are shared among the threads. Each thread receives its own copy of private variables. Any temporary intermediate-computation variables defined outside the loop must be private:

```
float x, y;


#pragma omp parallel for private(x,y)
for( int i = 0; i < num; i++ )
{
}
```

Variables that accumulate are especially critical. They can't be private, but they also must be handled carefully so they don't get out of sync.

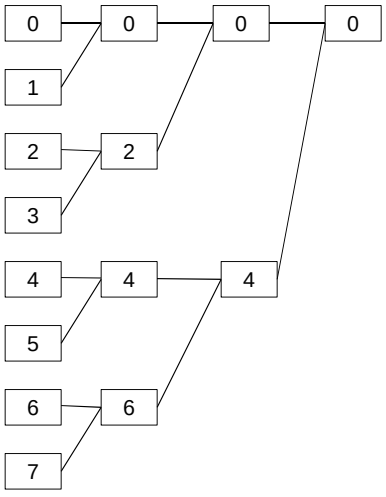
```
#pragma omp parallel for private(i,partialSum) reduction(+:total)
for( int i = 0; i < num; i++ )
{
        // compute a partial sum and add it to the total
        float partialSum = . . .
        total += partialSum;
}
```


In the reduction clause argument (A:B), A is the operation and B is the variable on which the operation will take place

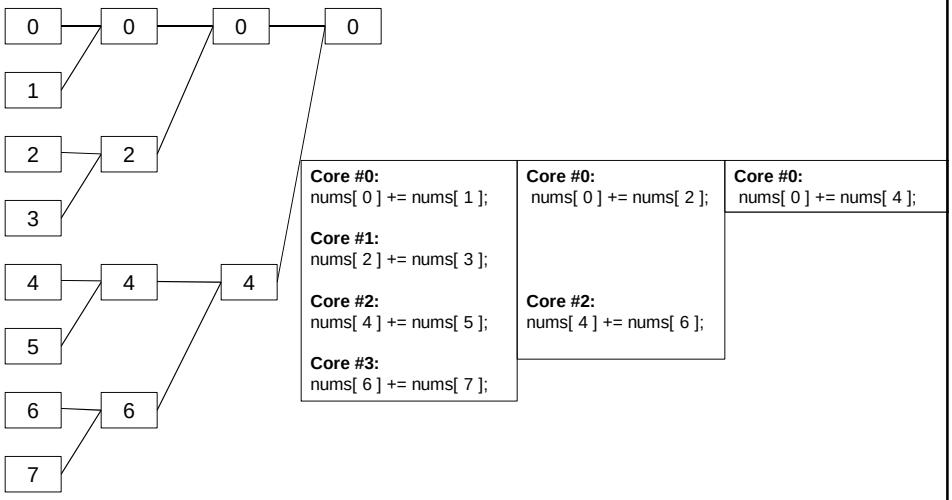
Here's How Reduction Really Works

Let's suppose we are adding up 8 numbers.

To do that, we will need 4 cores and 3 steps:



Here's How Reduction Really Works



Synchronizing OpenMP threads

The OpenMP paradigm is to create a mutual exclusion lock that only one thread can set at a time:

```
omp_lock_t Sync;
. . .
omp_init_lock( &Sync );
. . .

omp_set_lock( &Sync );
                << code that needs the mutual exclusion >>
omp_unset_lock( &Sync );


omp_test_lock( &Sync );
```

omp_set_lock blocks, waiting for the lock to become available

omp_test_lock does not block – this is good if there is some more computing that could be done if the lock is not yet available

Other OpenMP Operations

See if OpenMP is available:

```
#ifdef _OPENMP
. . .
#endif
```

Force all threads to wait until all threads have reached this point:

```
#pragma omp barrier
```

(Note: there is an implied barrier after parallel for loops and OpenMP sections, unless the *nowait* clause is used)

Make this operation atomic (i.e., cannot be split by thread-swapping):

```
#pragma omp atomic
      x += 5.;
```

(Note: this is important for read-modify-write operations like this one)