Parallel Programming: Background Information

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

Three Reasons to Study Parallel Programming

1. Increase performance: do more work in the same amount of time
2. Increase performance: take less time to do the same amount of work
3. Make some programming tasks more convenient to implement

Example:
Decrease the time to compute a simulation

Example:
Create a web browser where the tasks of monitoring the user interface, downloading text, and downloading multiple images is happening simultaneously

Example:
Increase the resolution, and thus the accuracy, of a simulation

Three Types of Parallelism:
1. Instruction Level Parallelism (ILP)

A program might consist of a continuous stream of assembly instructions, but it is not necessarily executed continuously. Oftentimes it has “pauses”, waiting for something to be ready so that it can proceed.

```
A = B + 1;
C = 3;
```

```
Load    B,r0
Add     $1,r0
Store   r0,A
Load    $3,r1
Store   r1,C
```

If B is not already in cache, this will ***block*** while B is fetched from memory

Out-of-order execution capability will slide instructions up if they can be executed while waiting for the block to end

```
Load    B,r0
Load    $3,r1
Store   r1,C
Load    B,r0
Add     $1,r0
Store   r0,A
```

If a compiler does this, it’s called ***Static ILP***
If the CPU chip does this, it’s called ***Dynamic ILP***

Three Types of Parallelism:
2. Data Level Parallelism (DLP)

Executing the same instructions on different parts of the data

```
for( i = 0; i < NUM; i++ )
{
        B[ i ] = sqrt(  A[ i ]  );
}
```

```
for( i = 0;  i  <  NUM/3;  i++ )
{
        B[ i ] = sqrt(  A[ i ]  );
}
```

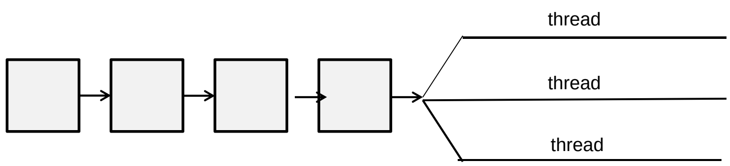
```
for( i = NUM/3;  i  <  2*NUM/3;  i++ )
{
        B[ i ] = sqrt(  A[ i ]  );
}
```

```
for( i = 2*NUM/3;  i  <  NUM;  i++ )
{
        B[ i ] = sqrt(  A[ i ]  );
}
```

Three Types of Parallelism:
3. Thread Level Parallelism (TLP)

Executing *different* instructions

Example: processing a variety of incoming transaction requests



In general, TLP implies that you have more threads than cores

Thread execution switches when a thread blocks or uses up its time slice

Flynn's Taxonomy

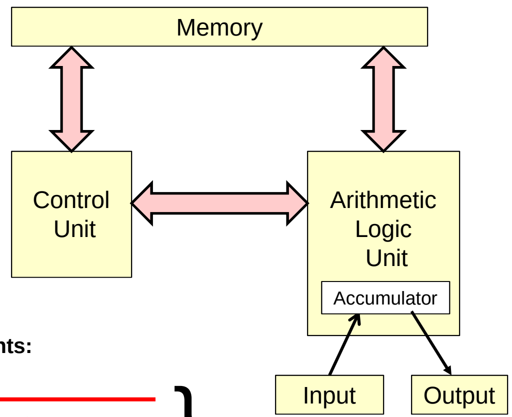
$$\left\{\frac{Single}{Multiple}\right\}\text{Instruction, }\left\{\frac{Single}{Multiple}\right\}\text{ Data}$$

Data →

Instructions ↓

SISD **"Normal" single-core CPU**	**SIMD** **GPUs, Special vector CPU instructions**
MISD **?????**	**MIMD** **Multiple processors running independently**

Von Neumann Architecture: Basically the fundamental pieces of a CPU have not changed since the 1960s



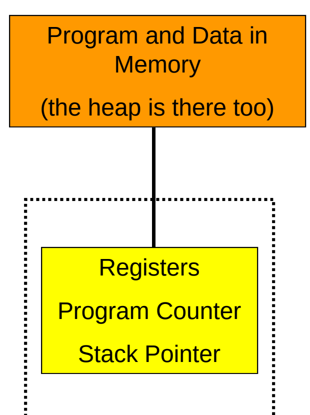
Other elements:

- Clock
- **Registers**
- **Program Counter**
- **Stack Pointer**

These together are the "state" of the processor

What Exactly is a Process?

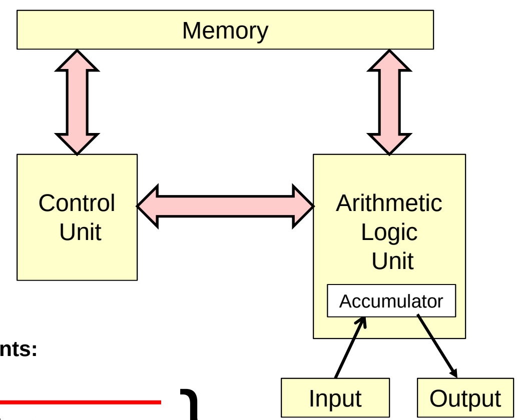
Processes execute a program in memory. The process keeps a state (program counter, registers, and stack).



Other elements:

- Clock
- **Registers**
- **Program Counter**
- **Stack Pointer**

Von Neumann Architecture: Basically the fundamental pieces of a CPU have not changed since the 1960s



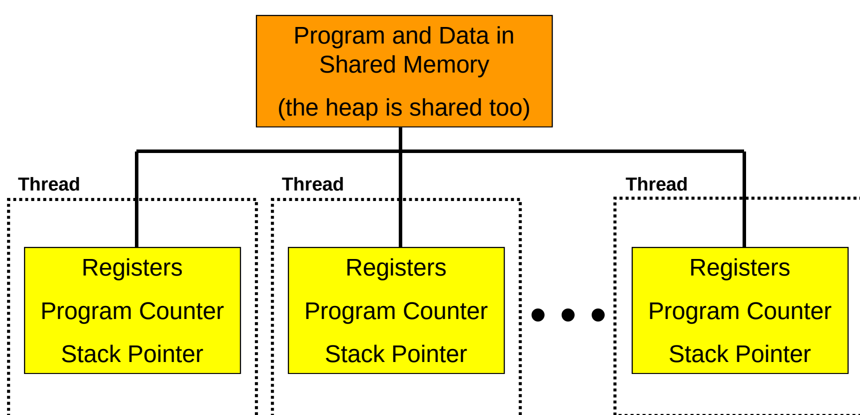
Other elements:

- Clock
- **Registers**
- **Program Counter**
- **Stack Pointer**

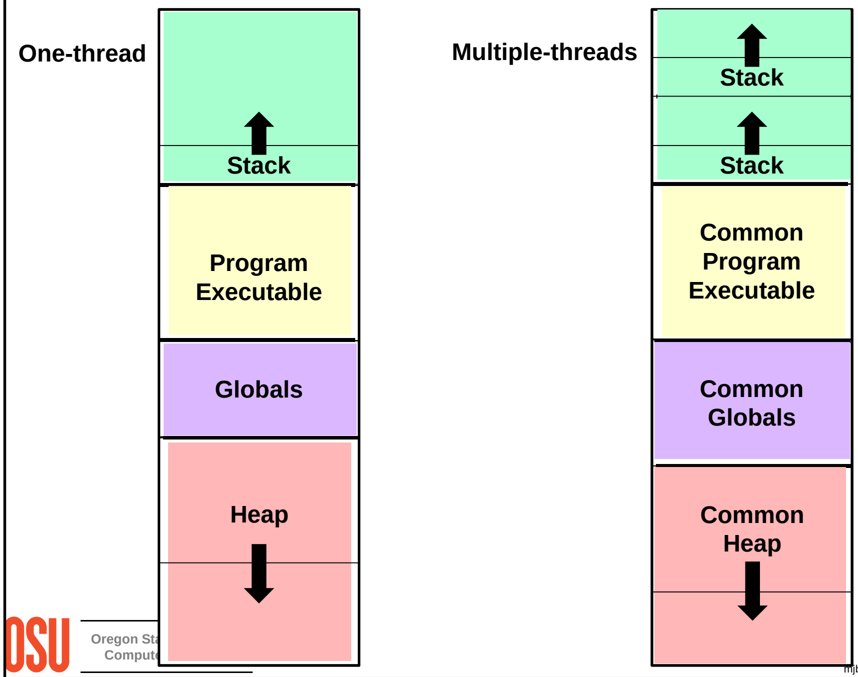
What if we include more than one set of these?

What Exactly is a Thread?

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



Memory Allocation in a Multithreaded Program



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 Pointer. But, since each thread is executing some part of the same program, each thread has access to the same global data in 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 – the *multicore* topic is coming soon!).

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
- Image loading
- 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 can make it easier to have many things going on in your program at one time, and can absorb the dead-time of other threads.

Watching out for Conflicts in Multithreaded Programs: Thread Safety

In order to use multithreading, one issue is that you must be sure your code is ***"thread-safe"*** (i.e., doesn't keep internal state between calls).

If you do keep internal state between calls, there is the chance that a second thread will pop in and change it, then the first thread will use it thinking it has not been changed.

Note that many of the standard C functions that we use all the time (e.g., *strtok*) are not thread safe:

```
char *strtok ( char * str,   const char * delims );
```

Watching out for Conflicts in Multithreaded Programs: Thread Safety

Thread #1

```
char *tok1 = strtok( Line1, DELIMS );   1

while( tok1 != NULL )
{
        . . .
        tok1 = strtok( NULL, DELIMS );   3
};
```

Execution Order

Thread #2

```
char *tok2 = strtok( Line2, DELIMS );   2

while( tok2 != NULL )
{
        . . .
        tok2 = strtok( NULL, DELIMS );
};
```

1. Thread #1 sets the internal character array pointer to somewhere in Line1[ ].

2. Thread #2 resets the internal character array pointer to somewhere in Line2[ ].

3. Thread #1 uses that internal character array pointer, but it is not pointing into Line1[ ] where Thread #1 thinks it left it.

Watching out for Conflicts in Multithreaded Programs: Thread Safety

Moral: if you will be multithreading, don't use internal static variables to retain state inside of functions.

In this case, using **strtok_r** is preferred:

```
char *strtok_r( char *str,  const char *delims,  char **sret );
```

strtok_r returns its internal state to you so that you can pass it back when you are ready. (The 'r' stands for "reentrant".)

Deadlock and Livelock Problems

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"

Worst of all, these problems are not always deterministic!

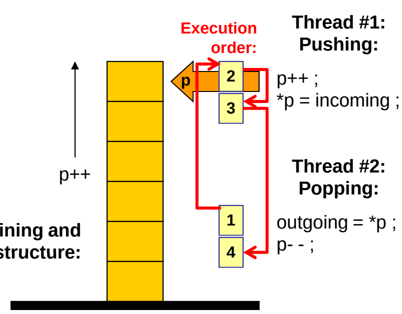
Race Condition Problems

Race Condition Fault

• A condition where it matters which thread gets to a particular piece of code first.

• Often comes about when one thread is modifying 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:



Worst of all, these problems are not always deterministic!

More Thread Safety: Locking Faults and Race Conditions

Deadlock and Livelock Faults

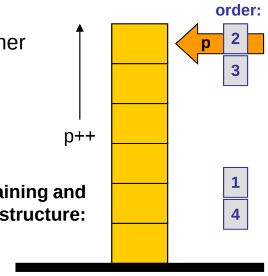
- *Deadlock*: Two threads are each waiting for the other to do something
- *Livelock*: like Deadlock, but both threads are changing state in sync with each other, possibly to avoid deadlock, and thus are still deadlocked

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:



Execution order:

Thread #1: Pushing:

p++ ;
*p = incoming ;

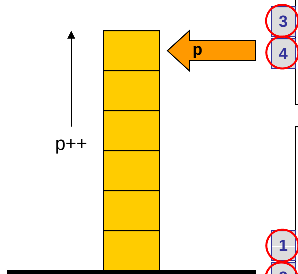
Thread #2: Popping:

outgoing = *p ;
p- - ;

Worst of all, these problems are not always deterministic!

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

Execution order:



Thread #1: Pushing:

```
. . .
Lock  A
{
3   p++ ;
4   *p = incoming ;
}
. . .
```

Thread #2: Popping:

```
. . .
Lock  A
{
1   outgoing = *p ;
2   p- - ;
}
. . .
```

Mutex Locks are usually named somehow so that you can have multiple ones with no ambiguity.

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

Sending a Message to the Optimizer: The *volatile* Keyword

The *volatile* keyword is used to let the compiler know that another thread might be changing a variable "in the background", so don't make any assumptions about what can be optimized away.

```
int val = 0;

        . . .

while(  val != 0  ) ;
```

A good compiler optimizer will *eliminate* this code because it "knows" that *val* == 0

```
volatile int val = 0;

        . . .

while(  val != 0  ) ;
```

The volatile keyword tells the compiler optimizer that it cannot count on *val* being == 0 here

Sending a Message to the Optimizer: The *restrict* Keyword

Remember our Instruction Level Parallelism example?

```
A = B + 1;
C = 3;
```

To assembly language

```
Load    B,r0
Add     $1,r0
Store   r0,A
Load    $3,r1
Store   r1,C
```

```
Load    B,r0
Load    $3,r1
Store   r1,C
Load    B,r0
Add     $1,r0
Store   r0,A
```

Optimize by moving two instructions up to execute while B is loading

Sending a Message to the Optimizer: The *restrict* Keyword

```
int *p;
int *q ;
   . . .
p = &B;
q = &B;
A = *p + 1;
*q = 3.;
```

Here the example has been changed slightly. This is what worries compilers, and keeps them from optimizing as much as they could.

Assembly language

```
Load    B,r0
Add     $1,r0
Store   r0,A
Load    $3,r1
Store   r1,B
```

```
Load    B,r0
Load    $3,r1
Store   r1,B
Load    B,r0
Add     $1,r0
Store   r0,A
```

"Optimized", but wrong, assembly language

Uh-oh! B is being loaded at the same time it is being stored into. Which value is correct?

Sending a Message to the Optimizer: The *restrict* Keyword

```
int * restrict p;
int * restrict q;
   . . .
A = *p + 1;
*q = 3.;
```

This is us promising that p and q will never point to the same memory location.

Assembly language

```
Load    B,r0
Add     $1,r0
Store   r0,A
Load    $3,r1
Store   r1,B
```

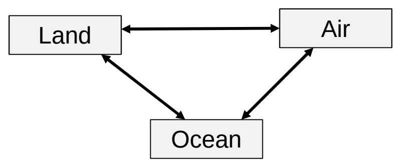
```
Load    B,r0
Load    $3,r1
Store   r1,B
Load    B,r0
Add     $1,r0
Store   r0,A
```

Optimized assembly language

Two Ways to Decompose your Problem into Parallelizable Pieces

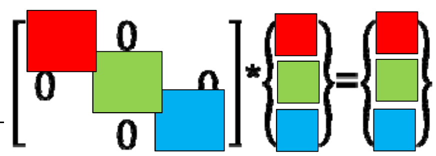
Functional (or Task) Decomposition

Breaking a task into sub-tasks that represent separate functions.
A web browser is a good example. So is a climate modeling program:



Domain (or Data) Decomposition

Breaking a task into sub-tasks that represent separate sections of the data. An example is a large diagonally-dominant matrix solution:

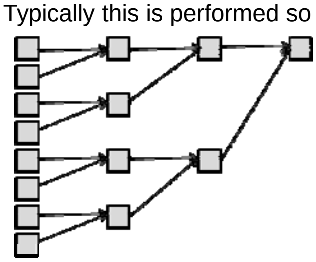


Some Definitions

Atomic An operation that takes place to completion with no chance of being interrupted by another thread

Deterministic The same set of inputs always gives the same outputs

Reduction Combining the results from multiple threads into a single sum or product, continuing to use multithreading. Typically this is performed so that it takes $O(\log_2 N)$ time instead of $O(N)$ time:



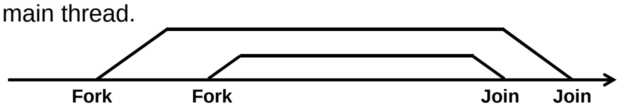
Fine-grained parallelism Breaking a task up into lots of small tasks

Coarse-grained parallelism Breaking a task up into a small number of large tasks

Barrier A point in the program where *all* threads must reach before *any* of them are allowed to proceed

Some More Definitions

Fork-join An operation where multiple threads are created from a main thread. All of those forked threads are expected to eventually finish and thus "join back up" with the main thread.



Shared variable After a fork operation, a variable which is shared among threads, i.e., has a single value

Private variable After a fork operation, a variable which has a private copy within each thread

Static Scheduling Dividing the total number of tasks T up so that each of N available threads has T/N sub-tasks to do

Dynamic scheduling Dividing the total number of tasks T up so that each of N available threads has *less than* T/N sub-tasks to do, and then doling out the remaining tasks to threads as they become available

Speed-up(N) T_1 / T_N
Speed-up Efficiency Speed-up(N) / N