Parallel Programming: Background Information

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
- Create a web browser where the tasks of monitoring the user interface, downloading text, and downloading multiple images are happening simultaneously
- 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.
   - Example:
     ```
     A = B + 1;
     C = 3;
     ```
   - 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
   - Example:
     ```
     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

2. Data Level Parallelism (DLP)
   - Executing the same instructions on different parts of the data
   - Example:
     ```
     for( i = 0; i < NUM; i++ )
     { B[i] = sqrt( A[i] ); }
     ```

3. Thread Level Parallelism (TLP)
   - Executing different instructions
   - Example: processing a variety of incoming transaction requests
   - Different Tasks
   - Thread execution switches when a thread blocks or uses up its time slice

Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>SISD</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Normal&quot; single-core CPU</td>
<td>GPUs, Special vector CPU instructions</td>
</tr>
<tr>
<td>MIMD</td>
<td>MIMD</td>
</tr>
<tr>
<td>Multiple processors running independently</td>
<td>Multiple processors running independently</td>
</tr>
</tbody>
</table>
### Von Neumann Architecture:

Basically the fundamental pieces of a CPU have not changed since the 1960s.

- **Memory**: The “Heap” (the result of a malloc or new call), is in here.
- **Control Unit**
- **Arithmetic Logic Unit**
- **Accumulator**
- **Input**
- **Output**

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).

- **Program and Data in Memory** (the heap is here too)
- **Registers**
- **Program Counter**
- **Stack Pointer**

### 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).

- **Program and Data in Shared Memory** (the heap is shared too)
- ** Registers**
- ** Program Counter**
- ** Stack Pointer**

### Memory Allocation in a Multithreaded Program

- **One-thread**
  - Stack
  - Program Executable
  -Globals
  - Heap

- **Multiple-threads**
  - Stack
  - Program Executable
  - Common Global
  - Common Heap

### 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.

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:

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

Watching out for Conflicts in Multithreaded Programs: Thread Safety

```
char *tok1 = strtok( Line1, DELIMS );
while( tok1 != NULL )
    { . . .
         tok1 = strtok( NULL, DELIMS );
    };

char *tok2 = strtok( Line2, DELIMS );
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.

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:

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

strtok_r returns its internal state to you so that you can store it locally and then pass it back when you are ready. (The 'r' stands for "re-entrant").

Deadlock Fault Problems

Deadlock Faults

Deadlock: Two threads are each waiting for the other to do something

Worst of all, the way these problems occur is not usually deterministic!

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, the way these problems occur is not usually deterministic!
Race Conditions can often be fixed through the use of Mutual Exclusion Locks (Mutexes)

Thread #1: Pushing:

```
Lock A
{
  p++;
  *p = incoming;
}
```

Thread #2: Popping:

```
Lock A
{
  outgoing = *p;
  p--;
}
```

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.

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

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

Here the example has been changed slightly. This is what worries the out-of-order mechanisms, and keeps them from optimizing as much as they could.

```
int *p;
int *q;
```

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

```
int * restrict p;
int * restrict q;
```

Now there is no conflict.

Sending a Message to the Optimizer: The restrict Keyword

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

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:

```
Land
```

```
 Air
```

```
Ocean
```

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:

```
0 0 0
0 0 0
0 0 0
```

What's really happening

What's really happening
Data Decomposition Reduces the Problem Size per Thread

Example: A diagonally-dominant matrix solution
- Break the problem into blocks
- Solve within the block
- Handle borders separately after a Barrier

\[
\begin{bmatrix}
0 & 0 \\
0 & 0 \\
\end{bmatrix} \ast \{ 0 \} = \{ 0 \}
\]

Barrier
Share results across boundaries

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 doing out the remaining tasks to threads as they become available.

Speed-up \(N\)

\[\text{Speed-up Efficiency} = \frac{T_1}{T_N} / N\]