Parallel Programming: Background Information and Tips

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 an existing simulation program

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

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

Two Types of Parallelism:
1. Data Level Parallelism (DLP)
   Threads are executing the same instructions on different 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] );
   }
   ```

Two Types of Parallelism:
2. Thread (or Task or Functional) Level Parallelism (TLP)
   Threads are executing different instructions

   Example: processing a variety of incoming transaction requests

   Different Tasks/Functions

   In TLP you can have more threads than cores

   Thread execution switches when a thread blocks or uses up its time slice
Flynn's Taxonomy

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>Single</td>
</tr>
<tr>
<td>Multiple</td>
<td>Multiple</td>
</tr>
</tbody>
</table>

- **SISD**
  - "Normal" single-core CPU
  - GPUs, Special vector CPU instructions

- **MISD**
  - Multiple processors running independently

- **SIMD**

- **MIMD**

*Normal* single-core CPU

GPUs, Special vector CPU instructions

Multiple processors running independently

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

The "Heap" (the result of a malloc or new call), is in here, along with Globals and the Stack

Control Unit

Arithmetic Logic Unit

Accumulator

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

Other elements:
- Clock
- Registers
- Program Counter
- Stack Pointer

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

The "Heap" (the result of a malloc or new call), is in here, along with Globals and the Stack

Control Unit

Arithmetic Logic Unit

Accumulator

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 pointer).

Program and Data in Shared Memory (the heap is shared too)

<table>
<thead>
<tr>
<th>Thread</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Registers</td>
</tr>
<tr>
<td>Program Counter</td>
<td>Program Counter</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>Stack Pointer</td>
</tr>
</tbody>
</table>

Program and Data in Shared Memory (the heap is shared too)

<table>
<thead>
<tr>
<th>Thread</th>
<th>Thread</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Registers</td>
<td>Registers</td>
</tr>
<tr>
<td>Program Counter</td>
<td>Program Counter</td>
<td>Program Counter</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>Stack Pointer</td>
<td>Stack Pointer</td>
</tr>
</tbody>
</table>

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

Memory Allocation in a Multithreaded Program

One-thread

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
</table>

Multiple-threads

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Executable</td>
</tr>
<tr>
<td>Globals</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Common Program Executable</td>
</tr>
<tr>
<td>Common Globals</td>
</tr>
<tr>
<td>Common Heap</td>
</tr>
</tbody>
</table>

Don’t take this completely literally. The exact arrangement depends on the operating system and the compiler. For example, sometimes the stack and heap are arranged so that they grow towards each other.

When is it Good to use Multithreading?

- When certain operations can become blocked, waiting for something else to happen
- When certain operations can be CPU-intensive
- When certain operations must respond to asynchronous I/O, including the user interface (UI)
- 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.
Some Definitions

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

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

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

Deterministic  The same set of inputs always gives the same outputs

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

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

Some More Definitions

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

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:

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

### Speed-up(N)

$T_1 / T_N$

### Speed-up Efficiency

$\frac{\text{Speed-up(N)}}{N}$

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

Parallel Programming Tips

Tip #1 -- Don’t Keep Internal State

```c
int GetLastPositiveNumber( int x )
{
    static int savedX;
    if( x >= 0 )
        savedX = x;
    return savedX;
}
```

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

Ironically, some of the standard C functions that we use all the time (e.g., `strtok`) keep internal state:

```c
char * strtok ( char * str, const char * delims );
```
Tip #1 -- Don't Keep Internal State

Thread #1

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

Thread #2

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 same 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.

Tip #1 -- Keep External State Instead

Thread #1

char * retValue1;
char * tok1 = strtok_r( Line1, DELIMS, &retValue1 );
while( tok1 != NULL )
{
    ... tok1 = strtok( NULL, DELIMS );
};

Thread #2

char * retValue2;
char * tok2 = strtok_r( Line2, DELIMS, &retValue2 );
while( tok2 != NULL )
{
    ... tok2 = strtok( NULL, DELIMS );
};

Now, execution order no longer matters!

Tip #1 -- Keep External State Instead

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 store it locally and then can pass it back when you are ready. (The ‘r’ stands for “re-entrant”.)

Tip #1 – Note that Keeping Global State is Just as Dangerous

Internal state:

    int GetLastPositiveNumber( int x )
    {
        static int savedX;
        if( x >= 0 )
            savedX = x;
        return savedX;
    }

Global state:

    int savedX;

    int GetLastPositiveNumber( int x )
    {
        int savedX;
        if( x >= 0 )
            savedX = x;
        return savedX;
    }

Tip #1 – Note that Keeping Global State is Just as Dangerous
Deadlock is when two threads are each waiting for the other to do something.

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

Tip #2 – Avoid Deadlock

A Race Condition is where it matters which thread gets to a particular piece of code first.

This often comes about when one thread is modifying a variable while the other thread is in the midst of using it.

A good example of a potential race condition situation is maintaining and using the pointer in a stack data structure:

Thread #1:

\[
\begin{align*}
\text{Pushing:} \\
p & \text{++} \\
*p & = \text{incoming} \\
p & \text{++}
\end{align*}
\]

Execution order:

Thread #2:

\[
\begin{align*}
\text{Popping:} \\
\text{outgoing} & = *p \\
p & \text{--}
\end{align*}
\]

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

Tip #3 – Avoid Race Conditions

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

Thread #1: Pushing:

\[
\begin{align*}
\text{...} \\
\text{MutexLock A} \\
\{ \\
p & \text{++} \\
*p & = \text{incoming} \\
\} \\
\text{...}
\end{align*}
\]

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

Thread #2: Popping:

\[
\begin{align*}
\text{...} \\
\text{MutexLock A} \\
\{ \\
\text{outgoing} & = *p \\
p & \text{--} \\
\} \\
\text{...}
\end{align*}
\]

We will talk about these a little later.

But note that, while solving a race condition, we can accidentally create a deadlock condition if the thread that owns the lock is waiting for the other thread to do something.

Tip #4 – 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;
\ldots
while( val != 0 );
```

A good compiler optimizer will eliminate this code because it “knows” that, for all time, val == 0

```
volatile int val = 0;
\ldots
while( val != 0 );
```

The volatile keyword tells the compiler optimizer that it cannot count on val being == 0 here.
Tip #5 – Beware of False Sharing Caching Issues

Note that using more threads initially gives a drop in performance!

We will get to this in the Caching notes!