# Table of Contents

1. Misc Information
2. Parallel Software: Developing & Writing
3. Next Time
HW 1 due today, 02/05/16
Quiz 1, 02/15/16, covers Module 1.
We will start Module 2 next week (MPI, Ch 3 of Pacheco)
Parallel Software for HPC

- Hardware and compilers continuously evolve
- Software must adapt to these changes
  - Compilers
  - Tool Libraries and API’s
  - Performance Profiling
  - Complexity abstraction (how to synchronize $10^5$ to $10^6$ processors?)
- Key issues in writing software:
  - Thread coordination
  - Shared memory
  - Distributed memory
Memory Distribution Patterns

- In shared memory programs:
  - Start a single process and fork threads.
  - Threads carry out tasks.

- In distributed memory programs:
  - Start multiple processes.
  - Processes carry out tasks.
A SPMD program consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

```
if ( I am thread process i )
    do something;
else
    do more interesting things;
```
Writing Parallel Programs

1. Divide the work among the processes/threads
   (a) so each process/thread gets roughly the same amount of work
   (b) and communication is minimized.

2. Arrange for the processes/threads to synchronize.

3. Arrange for communication among processes/threads.

```c
#include <stdio.h>

int main() {
    int i, n = 100000000;
    double x[n], y[n];
    ...
    for (i = 0; i < n; i++)
        x[i] += y[i];
    return 0;
}
```
Shared Memory

- Dynamic threads
  - Master thread waits for work, forks new threads, and when threads are done, they terminate
  - Efficient use of resources, but thread creation and termination is time consuming.

- Static threads
  - Pool of threads created and are allocated work, but do not terminate until cleanup.
  - Better performance, but potential waste of system resources.
Nondeterminism

```c
... printf ( "Thread %d > my_val = %d\n", my_rank, my_x ); ... 
```

Thread 0 > my_val = 7
Thread 1 > my_val = 19
Thread 1 > my_val = 19
Thread 0 > my_val = 7
Nondeterminism

```c
my_val = Compute_val ( my_rank );
x += my_val;
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Finish assignment to my_val</td>
<td>In call to Compute_val</td>
</tr>
<tr>
<td>1</td>
<td>Load x = 0 into register</td>
<td>Finish assignment to my_val</td>
</tr>
<tr>
<td>2</td>
<td>Load my_val = 7 into register</td>
<td>Load x = 0 into register</td>
</tr>
<tr>
<td>3</td>
<td>Add my_val = 7 to x</td>
<td>Load my_val = 19 into register</td>
</tr>
<tr>
<td>4</td>
<td>Store x = 7</td>
<td>Add my_val to x</td>
</tr>
<tr>
<td>5</td>
<td>Start other work</td>
<td>Store x = 19</td>
</tr>
</tbody>
</table>
Nondeterminism

- Race condition
- Critical section
- Mutually exclusive
- Mutual exclusion lock (mutex, or simply lock)

```c
my_val = Compute_val ( my_rank ) ;
Lock(&add_my_val_lock ) ;
x += my_val ;
Unlock(&add_my_val_lock ) ;
```
busy-waiting

my_val = Compute_val ( my_rank ) ;
if ( my_rank == 1 )
    while ( ! ok_for_1 ) ; /* Busy-wait loop */
x += my_val ; /* Critical section */
if ( my_rank == 0 )
    ok_for_1 = true ; /* Let thread 1 update x */
message-passing

char message [ 100 ];

...  
my_rank = Get_rank ( );
if ( my_rank == 1 ) {
    sprintf ( message , "Greetings from process 1" );
    Send ( message , MSG_CHAR , 100 , 0 );
} else if ( my_rank == 0 ) {
    Receive ( message , MSG_CHAR , 100 , 1 );
    printf ( "Process 0 > Received: %s\n" , message );
}
Partitioned Global Address Space Languages

```c
shared int n = ...;
shared double x[n], y[n];
private int i, my_first_element, my_last_element;
my_first_element = ...;
my_last_element = ...;
/* Initialize x and y */
... 
for (i = my_first_element; i <= my_last_element; i++)
    x[i] += y[i];
```
Input and Output

- In distributed memory programs, only process 0 will access *stdin*. In shared memory programs, only the master thread or thread 0 will access *stdin*.

- In both distributed memory and shared memory programs all the processes/threads can access *stdout* and *stderr*. 
Input and Output

- However, because of the indeterminacy of the order of output to `stdout`, in most cases only a single process/thread will be used for all output to `stdout` other than debugging output.

- Debug output should always include the rank or id of the process/thread that’s generating the output.
Input and Output

- Only a single process/thread will attempt to access any single file other than stdin, stdout, or stderr. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.
Next Time

- Next class: 02/08/16
- Q1, Monday 2/15/16
- Reading (see Links page for refs.)
  - Ch2, Pacheco 2011
  - Foster, Designing and Building Parallel Programs "An Introduction to Parallel Programming."
  - Ch3, MPI (Pacheco)