Table of Contents

1. Parallel Software: Developing & Writing
2. Parallel Software: Profiling Tools
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 programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

```c
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
double x[n], y[n];
...
for (i = 0; i < n; i++)
    x[i] += y[i];
```
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 0 > my_val = 7
Thread 1 > my_val = 19
```
**Nondeterminism**

```
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-blocking

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 [ 1 0 0 ] ;

...  

my_rank = Get_rank () ;

if ( my_rank == 1 ) {
    printf ( 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

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.
Simple profiling tools

- Unix *top* command
- PBS *qstat* command
- in-code timers (*C CPU_TIME()*)
- gprof (GNU Profile)
- integrated performance monitor (IPM)

Advanced profiling tools

- Allinea DDT
- Tuning and Analysis Utilities (TAU)
The output from the `top` command is shown below:

```
top - 16:38:31 up 5 days, 8:05, 7 users, load average: 0.28, 0.31, 0.20
Tasks: 176 total, 2 running, 174 sleeping, 0 stopped, 0 zombie
Cpu(s): 24.2%us, 0.8%sy, 0.0%ni, 73.5%id, 1.4%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 12188132k total, 4513528k used, 7674604k free, 29736k buffers
Swap: 33409020k total, 21692k used, 33387328k free, 1665928k cached

PID USER   PR  NI  VIRT  RES  SHR  %CPU %MEM    TIME+ COMMAND
16744 escalona 20   0 4664m 2.4g  896  R 99.7 20.9  0:11.82 histogram_mod
15234 hirakawa 20   0  105m 1812  1440 S  0.3  0.0  0:00.15 bash
16721 mthomas  20   0 15040 1292  940  R  0.3  0.0  0:00.05 top
   1 root      20   0  19364  696  480  S  0.0  0.0  0:02.61 init
   2 root      20   0    0    0    0  S  0.0  0.0  0:00.01 kthreadd
```
[mthomas@tuckoo hello]$ qsub batch.mpi-hello
40.tuckoo.sdsu.edu
[mthomas@tuckoo hello]$ qstat -a
[mthomas@tuckoo hello]$ qstat -a

tuckoo.sdsu.edu:

<table>
<thead>
<tr>
<th>Job ID</th>
<th>Username</th>
<th>Queue</th>
<th>Jobname</th>
<th>SessID</th>
<th>NDS</th>
<th>TSK</th>
<th>Memory</th>
<th>Time</th>
<th>S</th>
<th>Elap</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.tuckoo.sdsu.e</td>
<td>mthomas</td>
<td>batch</td>
<td>hello</td>
<td>22972</td>
<td>1</td>
<td>4</td>
<td>--</td>
<td>--</td>
<td>C</td>
<td>00:00</td>
</tr>
<tr>
<td>38.tuckoo.sdsu.e</td>
<td>mthomas</td>
<td>batch</td>
<td>mpi-hello</td>
<td>22560</td>
<td>2</td>
<td>8</td>
<td>--</td>
<td>--</td>
<td>C</td>
<td>00:00</td>
</tr>
<tr>
<td>39.tuckoo.sdsu.e</td>
<td>mthomas</td>
<td>batch</td>
<td>mpi-hello</td>
<td>22594</td>
<td>2</td>
<td>8</td>
<td>--</td>
<td>--</td>
<td>C</td>
<td>00:00</td>
</tr>
<tr>
<td>40.tuckoo.sdsu.e</td>
<td>mthomas</td>
<td>batch</td>
<td>mpi-hello</td>
<td>22630</td>
<td>2</td>
<td>8</td>
<td>--</td>
<td>--</td>
<td>C</td>
<td>00:00</td>
</tr>
</tbody>
</table>
Timing serial histogram code: gettimeofday()

```c
int main(int argc, char* argv[]) {
    int bin_count, i, bin;
    float min_meas, max_meas;
    float* bin_maxes;
    int* bin_counts;
    int data_count;
    float* data;

    gettimeofday(&tstart_wall, NULL);
    /* Check and get command line args */
    if (argc != 5) Usage(argv[0]);
    gettimeofday(&tstart_getargs, NULL);
    Get_args(argv, &bin_count, &min_meas, &max_meas, &data_count);
    gettimeofday(&tstop_getargs, NULL);

    /* Allocate arrays needed */
    gettimeofday(&tstart_mem, NULL);
    bin_maxes = malloc(bin_count*sizeof(float));
    bin_counts = malloc(bin_count*sizeof(int));
    data = malloc(data_count*sizeof(float));
    gettimeofday(&tstop_mem, NULL);

    /* Generate the data */
    gettimeofday(&tstart_gendat, NULL);
    Gen_data(min_meas, max_meas, data, data_count);
    gettimeofday(&tstop_gendat, NULL);

    /* Create bins for storing counts */
    gettimeofday(&tstart_genbins, NULL);
    Gen_bins(min_meas, max_meas, bin_maxes, bin_counts, bin_count);
    gettimeofday(&tstop_genbins, NULL);
}```
<table>
<thead>
<tr>
<th>ProbSize</th>
<th>$T_{wall}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>100</td>
<td>0.5</td>
</tr>
<tr>
<td>1000</td>
<td>0.6</td>
</tr>
<tr>
<td>10000</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Output from the Integrated Performance Monitor for jobs run on TACC resources.
Profiling the programs using GPROF

We can use profiling applications to analyze the program call tree and obtain some timings. How closely do our results agree?

```
PROFILING: using −p option in make
```

```
[mthomas@tuckoo ch2]$ mpicc −p −o histodat histodat.c
[mthomas@tuckoo ch2]$ ./histodat 10 1 1000 1000000
T.wall in seconds: 0.107674 seconds
.
T.whichbin in seconds: 0.085712 seconds
.
[mthomas@tuckoo ch2]$ gprof histodat gmon.out
% cumulative  self      self         total       name
 time  seconds  seconds  calls  ms/call  ms/call
   75.19    0.06     0.06  1000000 0.0000  0.0000 Which_bin
   12.53    0.07     0.01        1 10.0303 10.0303 Gen_data
   12.53    0.08     0.01        1 10.0303 10.0303 main
    0.00    0.08     0.00        1  0.0000  0.0000 Gen_bins
    0.00    0.08     0.00        1  0.0000  0.0000 Get_args

GPROF says that 75% of the time is spent in Which_bin, for 0.06 seconds. Using our Twall, we measured .086 seconds
```

Which approach is correct? GNU Profile: https://www.cs.utah.edu/dept/old/texinfo/as/gprof.html
Allinea DDT
TAU Performance System

[Image of TAU Performance System tool interface showing timeline and function summary for CUDA processes and threads.]