CSC510
Parallel Programming

Hardware and Software
(Part 3)

Dr. Joe Zhang

Contents

• Some background
• Modifications to the von Neumann model
• Parallel hardware
• Parallel software
• Input and output
• Performance
• Parallel program design
• Writing and running parallel programs
• Assumptions
PARALLEL SOFTWARE

The burden is on software

• Hardware and compilers can keep up the pace needed.
• From now on…
  – 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.

PP-3: Hardware and Software
SPMD – single program multiple data

- A SPMD (Single Program, Multiple Data) program consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

```plaintext
if (I’m thread process i)
    do this;
else
    do that;
```

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.
**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 are 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

PP-3: Hardware and Software
Nondeterminism

\[ \text{my\_val} = \text{Compute\_val (my\_rank)}; \]
\[ x += \text{my\_val}; \]

<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>

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

\[ \text{my\_val} = \text{Compute\_val (my\_rank)}; \]
\[ \text{Lock(&add\_my\_val\_lock)}; \]
\[ x += \text{my\_val}; \]
\[ \text{Unlock(&add\_my\_val\_lock)}; \]
**busy-waiting**

\[
\text{my_val} = \text{Compute_val ( my_rank ) ;}
\]
\[
\text{if ( my_rank == 1)}
\]
\[
\quad \text{while (! ok_for_1 ) ; } /* \text{Busy-wait loop */}
\]
\[
x += \text{my_val } ; /* \text{Critical section */}
\]
\[
\text{if ( my_rank == 0)}
\]
\[
\quad \text{ok_for_1 } = \text{true ; } /* \text{Let thread 1 update } x */
\]

**message-passing**

\[
\text{char message }[100] ;
\]
\[
\ldots
\]
\[
\text{my_rank } = \text{Get_rank }() ;
\]
\[
\text{if ( my_rank } == 1) \{ \\
\quad \text{sprintf ( message , "Greetings from process 1" ) ;} \\
\quad \text{Send ( message , MSG_CHAR , 100 , 0 ) ;} \\
\} \text{ else if ( my_rank } == 0) \{ \\
\quad \text{Receive ( message , MSG_CHAR , 100 , 1 ) ;} \\
\quad \text{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.

- 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.
### Speedup

- Number of cores = \( p \)
- Serial run-time = \( T_{\text{serial}} \)
- Parallel run-time = \( T_{\text{parallel}} \)

\[
T_{\text{parallel}} = T_{\text{serial}} / p
\]

**Speedup of a parallel program**

\[
S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}
\]
Efficiency of a parallel program

\[ E = \frac{S}{p} = \frac{\frac{T_{\text{serial}}}{T_{\text{parallel}}}}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} \]

Speedups and efficiencies of a parallel program

<table>
<thead>
<tr>
<th>( p )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>1.0</td>
<td>1.9</td>
<td>3.6</td>
<td>6.5</td>
<td>10.8</td>
</tr>
<tr>
<td>( E = S/p )</td>
<td>1.0</td>
<td>0.95</td>
<td>0.90</td>
<td>0.81</td>
<td>0.68</td>
</tr>
</tbody>
</table>
## Speedups and efficiencies of parallel program on different problem sizes

<table>
<thead>
<tr>
<th></th>
<th>$p$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>$S$</td>
<td>1.0</td>
<td>1.9</td>
<td>3.1</td>
<td>4.8</td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td>$E$</td>
<td>1.0</td>
<td>0.95</td>
<td>0.78</td>
<td>0.60</td>
<td>0.39</td>
</tr>
<tr>
<td>Original</td>
<td>$S$</td>
<td>1.0</td>
<td>1.9</td>
<td>3.6</td>
<td>6.5</td>
<td>10.8</td>
</tr>
<tr>
<td></td>
<td>$E$</td>
<td>1.0</td>
<td>0.95</td>
<td>0.90</td>
<td>0.81</td>
<td>0.68</td>
</tr>
<tr>
<td>Double</td>
<td>$S$</td>
<td>1.0</td>
<td>1.9</td>
<td>3.9</td>
<td>7.5</td>
<td>14.2</td>
</tr>
<tr>
<td></td>
<td>$E$</td>
<td>1.0</td>
<td>0.95</td>
<td>0.98</td>
<td>0.94</td>
<td>0.89</td>
</tr>
</tbody>
</table>

### Speedup

![Speedup Graph]

**PP-3: Hardware and Software**
Effect of overhead

\[ T_{\text{parallel}} = \frac{T_{\text{serial}}}{p} + T_{\text{overhead}} \]

PP-3: Hardware and Software
Amdahl’s Law

• Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.

Example

• We can parallelize 90% of a serial program.
• Parallelization is “perfect” regardless of the number of cores $p$ we use.
• $T_{\text{serial}} = 20$ seconds
• Runtime of parallelizable part is

$$0.9 \times T_{\text{serial}} / p = 18 / p$$
Example (cont.)

• Runtime of “unparallelizable” part is

\[ 0.1 \times T_{\text{serial}} = 2 \]

• Overall parallel run-time is

\[ T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18 / p + 2 \]

• Speed up

\[ S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18 / p + 2} \]

Scalability

• In general, a problem is scalable if it can handle ever increasing problem sizes.
• If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is strongly scalable.
• If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is weakly scalable.
Taking Timings

- What is time?
- Start to finish?
- A program segment of interest?
- CPU time?
- Wall clock time?

```c
double start, finish;
... 
start = Get_current_time();
/* Code that we want to time */
... 
finish = Get_current_time();
printf("The elapsed time = %.3f seconds\n", finish-start);
```

MPI_Wtime
omp_get_wtime
Taking Timings

```c
private double start, finish;
...
start = Get_current_time();
/* Code that we want to time */
...
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

Taking Timings

```c
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
...
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();
/* Code that we want to time */
...
my_finish = Get_current_time();
my_elapsed = my_finish - my_start;
/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
    printf("The elapsed time = %e seconds\n", global_elapsed);
```
Foster’s methodology

1. **Partitioning**: divide the computation to be performed and the data operated on by the computation into small tasks.

   The focus here should be on identifying tasks that can be executed in parallel.

2. **Communication**: determine what communication needs to be carried out among the tasks identified in the previous step.
Foster’s methodology

3. **Agglomeration or aggregation**: combine tasks and communications identified in the first step into larger tasks.

For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

4. **Mapping**: assign the composite tasks identified in the previous step to processes/threads.

This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

---

Example - histogram

- 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9
Serial program - input

1. The number of measurements:  \texttt{data\_count}
2. An array of  \texttt{data\_count} floats:  \texttt{data}
3. The minimum value for the bin containing the smallest values: \texttt{min\_meas}
4. The maximum value for the bin containing the largest values: \texttt{max\_meas}
5. The number of bins: \texttt{bin\_count}

Serial program - output

1. \texttt{bin\_maxes} : an array of \texttt{bin\_count} floats
2. \texttt{bin\_counts} : an array of \texttt{bin\_count} ints
First two stages of Foster’s Methodology

**Find_bin**

\[
\text{data}[i-1] \quad \text{data}[i] \quad \text{data}[i+1]
\]

**Increment**

\[
\text{bin_counts}[b-1]++ \quad \text{bin_counts}[b]++
\]

PP-3: Hardware and Software

Alternative definition of tasks and communication

**Find_bin**

\[
\text{data}[i-1] \quad \text{data}[i] \quad \text{data}[i+1] \quad \text{data}[i+2]
\]

\[
\text{loc_bin}_{\text{cts}}[b-1]++ \quad \text{loc_bin}_{\text{cts}}[b]++
\]

\[
\text{loc_bin}_{\text{cts}}[b-1]++ \quad \text{loc_bin}_{\text{cts}}[b]++
\]

\[
\text{bin_counts}[b-1]+= \quad \text{bin_counts}[b]+=\n\]

PP-3: Hardware and Software
Concluding Remarks (1)

• Serial systems
  – The standard model of computer hardware has been the von Neumann architecture.

• Parallel hardware
  – Flynn’s taxonomy.

• Parallel software
  – We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching.
    – SPMD programs.
Concluding Remarks (2)

• Input and Output
  – We’ll write programs in which one process or thread can access stdin, and all processes can access stdout and stderr.
  – However, because of nondeterminism, except for debug output we’ll usually have a single process or thread accessing stdout.

Concluding Remarks (3)

• Performance
  – Speedup
  – Efficiency
  – Amdahl’s law
  – Scalability
• Parallel Program Design
  – Foster’s methodology
CSC510
Parallel Programming

Questions?

Dr. Joe Zhang
PP_3: Hardware and Software