Programming parallel computing systems

Program
(using some programming model)

Compiler

Parallel processes (threads) + Access to address space

Run-time system

Parallel architecture
(Multiple processors and a physical memory architecture)

Note the decoupling between the programming model and the physical architecture – For instance, a parallel program can run on a single processor!!!
Two schools for programming parallel systems:

- Automatic detection of parallelism in serial programs and automatic distribution of data and computation.
- User specified parallelism (data distribution, computation distribution, or both).

Writing parallel programs:

1. Divide the work among the processes/threads such that
   (a) each process/thread gets roughly the same amount of work
   (b) communication is minimized.
2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among processes/threads.

Parallel Programming Models (control threads - processes).

1) Start with one control thread, and create other threads when needed
   Examples: Pthreads (explicit thread creation) and OpenMP (implicit thread creation).

2) Start with multiple control threads – usually multiple copies of the same program (SPMD – single program, multiple data).
Parallel Programming Models (scope of variables).

1) Variables declared shared among threads or processes – any process can read/write to these variables.

Problems with race conditions???

2) Variables declared private to a process or thread

To make the value of a private variable available to other processes, one has to either exchange messages, or copy the value to a shared variable.

A programming model can combine private and shared variables, as well as allow message passing.

Thread creation strategies

- On demand, Dynamic thread creation
  - 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 thread creation
  - Pool of threads created and are allocated work, but do not terminate until cleanup.
  - Better performance, but potential waste of system resources.
Example - Pthreads

```c
int main(int argc, char *argv) {
    double A[100];  /* global, shared variable*/
    int i;
    ...
    for (i = 0; i < 4; i++)  pthread_create( … , DoStuff, int i ) ;
    … /* execution continues in parallel with 4 copies of DoStuff*/
    …
    for (i = 0; i < 4; i++)  pthread_join (… , DoStuff, …) ;
    …
}

void DoStuff (int threadID) {
    int k; /* k is a local variable – each instance of DoStuff has a copy*/
    … /* do stuff in parallel with main */
    for (k = threadID*25 ; k < (threadID+1)*25 ; k++) … do something with A[k] …
    …
}
```

The five threads can be executed on separate CPUs or time_shared on one CPU.

Example - OpenMP

```c
int main(){
    print("Start\n");
    … /* serial code */
    #pragma omp parallel {
        …
        printf("Hello World\n");
        …
    }
    … /* resume serial code */
    printf("Done\n");
}
```

% Result of execution
Start
Hello World
Hello World
Hello World
Hello World
Done

The user can control the number of parallel threads by setting the environment variable setenv OMP_NUM_THREADS 4
Example - OpenMP

```c
#define n 1000
int main()
int i, a[n], b[n], c[n];
...

#pragma omp for shared(a,b,c), private(i)
{ for (i = 0; i < n; i++)
c[i] = a[i] + b[i];
} /* end of parallel section */
...
/* resume serial code */
```

The loop will be automatically broken down into smaller loops and each small loop will be given to one thread.

Warning: the loop iterations should be independent (no loop carried dependences)

Example – a message passing program

```c
int main()
int x, sum, i; /* local variables */
...
call a function to get the num_processors;
...
call a function to get your processorID;
compute a local value for x;
if (processorID > 0)
    send the value of x to processor 0;
else {
    sum = x;
    for (i = 1; i < num_processors; i++)
        receive a value from processor i;
        add that value to sum
}
}
```

In SPMD, the number of processes (threads) is specified before execution starts.

```
processID = 0 1 2 3
x = 10 20 40 80
```

```
sum = 10
sum = 30
sum = 70
sum = 150
```

```
sum = 150 ?? ?? ??
```
Avoiding race conditions

- To guarantee correctness, use critical sections
- Enforce mutual exclusion
- Can use mutual exclusion lock (mutex, or simply lock)

```c
my_val = Compute_val();
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

Busy-waiting to enforce order

/* Initially, ok_for_1 = 0 */

```c
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 */
```

How do you extend this method to more than two threads?
Input and Output

- Only one thread/process should access stdin.

- All processes/threads may access stdout, but it is clearer if only one process/thread accesses stdout.

- 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
Speedup, S, and Efficiency, E.

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

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

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

S and E change with problem sizes

<table>
<thead>
<tr>
<th>Problem Size</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></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></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></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></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></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></td>
<td>1.0</td>
<td>0.95</td>
<td>0.98</td>
<td>0.94</td>
<td>0.89</td>
</tr>
</tbody>
</table>
**Amdahl’s Law**

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

Let $\alpha$ be the fraction of a program that has to be performed serially, then, using $p$ processors, the maximum possible speedup is:

$$ S < \frac{1}{\alpha + \frac{(1-f)}{p}} $$

Hence, even with unlimited number of processors, the speedup cannot be larger than $1/\alpha$.

**Example**

- We can parallelize 90% of a serial program.
- Parallelization is “perfect” for any number of cores
- $T_{\text{serial}} = 20$ seconds
- 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?
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);
```

Need to find the maximum across all threads

PARALLEL PROGRAM DESIGN
Foster’s methodology

1. **Partitioning**: divide the computation and the data operated on by the computation into small tasks.
2. **Communication**: determine what communication needs to be carried out among the tasks.
3. **Aggregation**: combine tasks and communications into larger tasks.
4. **Mapping**: assign the composite tasks identified in the previous step to processes/threads.

Goal: balance the load and minimize communication.

---

**Example - histogram**

- **data** = 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

For each bin, find the number of data elements and the maximum data value.
First two stages of Foster’s Methodology

---

Find bin: \[ \text{data}[i-1] \quad \text{data}[i] \quad \text{data}[i+1] \ldots \]

Increment bin counts: \[ \text{bin counts}[b-1]++ \quad \text{bin counts}[b]++ \ldots \]

Find bin: \[ \ldots \quad \text{loc bin cts}[b-1]++ \quad \text{loc bin cts}[b]++ \quad \ldots \]

---

Copyright © 2010, Elsevier Inc. All rights Reserved