CIS 631: Parallel Processing

Dr. Allen D. Malony

Midterm – May 22, 2003

NAME: __________________________

<table>
<thead>
<tr>
<th>Section</th>
<th>Total Points</th>
<th>Points Scored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Parallel Programming</td>
<td>5, 5, 5, 15, +10</td>
<td>/ 30</td>
</tr>
<tr>
<td>2. Parallel Algorithms</td>
<td>20, 15</td>
<td>/ 35</td>
</tr>
<tr>
<td>3. Performance Analysis</td>
<td>5, 5, 5, 25</td>
<td>/ 45</td>
</tr>
<tr>
<td>Total</td>
<td>110</td>
<td>/ 110</td>
</tr>
</tbody>
</table>

Comments:

1. Sections 1.1 and 3.1 are short answers questions.

2. Sections 1.2 and 2.1 may take some time.

3. Section 3.2 is probably the hardest. Don’t worry if you don’t have a calculator. As long as the formulas are correct, you can give approximate answers, as best you can, when asked for numbers.

4. Section 1.3 is extra credit. It is recommended that you not spend any time on it until you are done with everything else.

1 Parallel Programming (30+10)

1.1 Short Answer (5,5,5)

a) Explain what is meant by SPMD. Given three reasons why SPMD is useful or interesting as a programming model?
Answer. SPMD stands for Single-Program Multiple-Data. It is a style of parallel programming that allows you to develop one version of the source code (a single program) which will then run on multiple processes on the parallel system. In the code, you distinguish between those processes by process identifiers (PIDs). The processes are logically assigned different (multiple) parts of the program data to compute on. When data is needed from other processes, they must communicate in some way to get the data. Usually, this is done with message passing. So, three things that are useful or interesting are:

1. Only one program is written.
2. SPMD programs are generally easy to scale.
3. SPMD tend to be portable between parallel machines.

There are other things useful or interesting in addition to these.

b) What are the benefits of non-blocking message communication? Why not use non-blocking message communication all the time?

Answer. Non-blocking message communication allows a sender or receiver to perform a send or receive operation, but not be required to wait for its completion. The sending or receiving process can continue executing while the communication is being performed. The ability to overlap communication and computation is the major benefit. However, the process must at some time later check to see if the communication completed and finish the operation. If there is more computation to do, the process can just check. Otherwise, it must wait for the communication operation to complete.

Non-blocking message communication should not be used all the time because there is some overhead with using it. When there is little advantage of overlapping communication and computation, blocking communication may be more efficient.

c) In shared memory parallel programming, does it ever make sense to create more threads than the number of available processors to use in a computation? Explain why or why not.

Answer. Certainly. You gain advantages from threading just like you would on a single processor computer system. One scenario where it makes a lot of sense is when there is I/O being performed. Another is to have one thread performing communication while another is executing. Generally, when there is available concurrency (so that additional threads have something to do) AND the threads could get blocked on other activities during their execution (e.g., I/O), creating more threads than available processors may be worthwhile.
1.2 Task Scheduling (15)

Consider the following nested loops:

\[
\begin{align*}
\text{do } & i=1,n \\
\text{do } & j=1,i \\
& a[i][j] = f(i,j) \\
& \text{end} \\
& \text{end}
\end{align*}
\]

Suppose the loop is executed on a shared-memory machine using multi-threading. You can either parallelize the outer loop or the inner loop. Which would you choose and why? You can assume there are no side-effects to the array as a result of the function call.

**Answer.** It is almost always better to parallelize the outer loop of nested loops since it increases the parallelism granularity.

Assume the assignment statement takes 3 time units to execute. Assume that the iterations of the loop being parallelized are scheduled dynamically among the threads, but only one iteration can be assigned at a time. The overhead for parallel iteration scheduling is 2 time units. The time to execute the “do” statement of the sequential loop (i.e., the time needed to update the iteration index) is 1 time unit. Let \(n = 8\). There are four threads available.

For each parallelization strategy, draw the activity of each thread on the time plots in Figure 1, using the values above. Distinguish (in some way, your choice) between activity states for each thread (e.g., idle, assignment, increment index, parallel iteration schedule, and so on). Be sure to show the period of time each state is active. Assume all threads are ready at time 0.

**Answer.** Parallelizing the outer loop will incur a parallel scheduling cost per iteration. Each thread then executes the inner loop independent of the other threads. Parallelizing the inner loop means that the outer loop will be executed sequentially. The inner loop scheduling will incur the parallel scheduling cost.
Parallelize Outer Loop

Parallelize Inner Loop

Figure 1: Task Scheduling Timeline.
1.3 OpenMP and Pthreads (Extra credit, +10)

OpenMP compilers work by interpreting the OpenMP directives in the program and generating a new multi-threaded program. This program interfaces with the OpenMP runtime library during parallel execution to implement the OpenMP directives used. You can think of an OpenMP compiler functioning as a source-to-source translator, rewriting the sequential program (plus OpenMP directives) to create threads of execution and control their operation. Certainly, compilers can perform sophisticated transformations on the program to make it run more efficiently, but the translation of basic OpenMP constructs is relatively straightforward.

Pretend you are an OpenMP compiler. Figure 2 below shows a skeleton of an OpenMP program in the top box and a partial translation of the program in the lower box. Fill in the white boxes with the appropriate thread code to implement the desired OpenMP functionality. Please show real code (to the extent possible) and not pseudo code.

```c
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

```c
int a, b;
main() {
    // serial segment
    for (i=0; i<8; i++)
        pthread_create(thread[i], attribute, function, arguments);
    for (i=0; i<8; i++)
        pthread_join(thread[i], NULL);
    // rest of serial segment
}
void *function(void *argument) {
    int a;
    // parallel segment
}
```

Figure 2: OpenMP Translation to Multi-threaded Program.
2 Parallel Algorithms (35)

2.1 Task Graphs (20)

A task graph shows the tasks that need to be performed in a computation (the nodes in the graph) and the execution ordering relationships between tasks (represented by the edges). A task cannot begin executing until the task(s) immediately incident to the task node (i.e., tasks with an edge directly to the task node) have completed. The task graph may denote the time needed to execute each task by a number next to the task node; otherwise, it is assumed to take unit time.

For each of the task graphs shown in Figures 3, determine the following:

- maximum degree of concurrency
- critical path length
- maximum achievable speedup assuming an infinite number of processors
- minimum number of processes needed to obtain the maximum speedup
- maximum speedup if the number of processes is limited to 4

Figure 3: Task Graphs.
Answer. Graph (a)

max degree concurrency 8
critical path length 22
max speedup \((3+7+5+2+5+3+5+8+7+4+5+6+5+8)/22 = 78/22 = 3.5\)
min processes 6
max speedup w/ four 78/28 = 2.8
max degree concurrency 8
critical path length 25
max speedup \( (5+2+5+8+2+4+6+1+3+7+1+5+5+7+5)/25 = 66/25 = 2.6 \)
min processes 4
max speedup w/ four 66/25 = 2.6
Graph (c)

```
1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
1 ..........*
2 +..........
3 +................
4 +..........
5 +....
6 +..........
7 +..........
8 +.*
9 +................
A +...*
B +.....*
C +.*
D +...*
E +.....
```

```
1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
1 1111111111188AAAABBBBBBCCCDDDEEEEEE
2 333333333333333366666666666666
3 99999999999999997777777777777777
4 222222222244444444445555
```

max degree concurrency 8
critical path length 17
max speedup \((5+5+8+5+2+5+5+1+8+2+3+1+2+3)/17 = 55/17 = 3.2\)
min processes 5
max speedup w/ four 55/18 = 3.0
Graph (d)

1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
1 ............*
2 +........................................
3 +........
4 +........................
5 +........*
6 +........
7 +........
8 +........
9 +........
A +........
B +........
C +........
D +........
E +........
F +...

max degree concurrency 4
critical path length 37
max speedup \((5+20+5+12+5+7+5+10+4+5+4+5+4+5+1)/37 = 97/37 = 2.6\)
min processes 4
max speedup w/ four 97/37 = 2.6
2.2 Component Labeling (15)

The component labeling problem in image processing is defined as follows. We are given a two-dimensional array of pixels valued 0 or 1. The 1-pixels must be labeled in such a way that two pixels have the same label if and only if they are in the same connected component. Two 1-pixels are in the same connected component if there is a path of contiguous 1-pixels linking them together. Two pixels are contiguous if they are adjacent vertically or horizontally. Sketch out a parallel algorithm to solve the component labeling problem assuming that you have a task for each pixel. (Hint: First, assign each 1-pixel a unique label.)

Answer.

1. Assume that the two-dimensional pixel array is organized as a \( nxn \) array. The first step is to distribute the array to all \( nxn \) tasks. This could be done in several ways. Although not very efficient, we’ll assume that the master process broadcasts the entire pixel array to every task and each task then extracts its associated pixel to work with.

2. Now we use the hint and set each 1-pixel to a unique label. Let this label be computed by the function \( l_{i,j} = n \ast i + j + 1 \). This operation can be done entirely in parallel for all tasks holding a 1-pixel.

3. To find out which neighbors have a 1-pixel, every task does the following:
   (a) send its label to each of the four neighbors (0 is sent if the task has a 0-pixel)
   (b) receive labels from each of the four neighbors
   (c) record which neighbors have a 1-pixel

4. For tasks with a 1-pixel, do the following:
   (a) compute the minimum of the 1-pixel labels obtained from last communication
   (b) the task’s new label is this minimum label ( \( l_{i,j} = \min(l_{i-1,j}, l_{i+1,j}, l_{i,j-1}, l_{i,j+1}) \))
   (c) send the new label to each of the 1-pixel neighbors
   (d) receive labels from each of the 1-pixel neighbors

5. Repeat Step 4 for \( n^2 - 2 \) times.

6. At this point, all 1-pixels of each connected component have been assigned a label that is the same within the component, but different from all other connected components. Now, we need to communicate the pixel array back to the master process.
3 Performance Analysis (45)

3.1 Short Answer (5,5,5,5)

a) In the context of parallel computing, “Amdahl’s Law” is regarded as applying to computations of “fixed” size. Define Amdahl’s Law respect to this concept.

**Answer.** Amdahl’s Law compares the time to solve a sequential problem to the time to solve this problem in parallel. We can write the runtime of the sequential computation as \( W = T_{seq} + T_{par} \), where \( T_{seq} \) is the part that cannot be parallelized and \( T_{par} \) is the sequential execution time part that can be parallelized. From this, we can compute the parallel runtime to be \( T_p = T_{seq} + \frac{T_{par}}{p} \), where \( p \) is the number of processors used in the parallel execution. We can rewrite \( T_{seq} \) as \( T_{seq} = f \times W \), where \( f \) is the sequential time fraction of \( W \). Then

\[
T_p = T_{seq} + \frac{W - T_{seq}}{p} = f \times W + \frac{W - f \times W}{p} \tag{1}
\]

Now, if we solve for speedup, \( S = W/T_p \), we get

\[
S = \frac{W}{f \times W + \frac{W - f \times W}{p}} = \frac{1}{f + \frac{1 - f}{p}} \tag{2}
\]

What we notice is that Amdahl’s Law depends entirely on the non-parallelizable (serial) fraction of the sequential problem. The amount of work represented by the sequential problem to be done, and hence the serial fraction, does not change in the parallel execution. This is why Amdahl’s Law is regarded as applying to computations of “fixed” size.

b) Given two algorithms, A and B, whose parallel speedups are \( S_p^A \) and \( S_p^B \), respectively, under what conditions can you claim A is a “faster” algorithm than B?

**Answer.** You cannot say which algorithm will run faster based solely on the speedups. Thus, the simplest condition is that A’s execution time is less than B’s. In this case, we make no claims regarding the relationship between A and B.

If A and B are solving the same problem, then A would be faster than B in those cases where \( S_p^A > S_p^B \), for the same choice of \( p \). Why? Because we are computing \( S_p^A \) and \( S_p^B \) using the same sequential algorithm, in fact, that fastest sequential algorithm we know. Thus, we assume a sequential time of 1, and reduce the speedup relationship to a time relationship. Notice, however, that the “faster” relationship here is dependent on \( p \).

c) List three scenarios where superlinear speedup could occur.
1. Data too large to fit in cache in sequential execution.

2. Problem is a search problem where all paths can be searched in parallel.

3. Problem involves I/O that is done asynchronously to computation.

d) For a given problem size, why does the efficiency go down as the number of processing elements increase? Is this always true?

**Answer.** Efficiency generally goes down because overhead of parallel computation increases with more processing elements. It is not true that efficiency decreases because there may be some execution effects that occur during sequential execution that do not occur in the same way in parallel execution, leading to improved performance that offsets extra parallel overhead (e.g., see superlinear scenarios above.)

### 3.2 Speedup, Efficiency, and Isoefficiency (25)

Consider the problem of computing the dot product of two vectors, $A$ and $B$, each of length $n$. The dot product is defined as:

\[
A \cdot B = \sum_{i=1}^{n} a_i b_i , 1 \leq i \leq n
\]  

(3)

**a)** Describe how you would parallelize this problem.

**Answer.** This problem is very similar to the problem of adding $n$ numbers, except here we have a multiplication step beforehand. The approach would be as follows:

1. Partition the vectors equally across the available $p$ processing elements and distribute partitions from master process. $O(n)$ communication time.

2. Compute $a_i \times b_i$ for all $n/p$ elements assigned to a processor. This can be done in parallel across all processors. $O(n/p)$ multiplication time.

3. Add all $n/p$ element products computed in 2) on each processor. This can be done in parallel across all processors. $O(n/p)$ addition time.

4. Sum all partial sums computed on each processor using a logical binary tree addition algorithm. $O(\log p)$ communication and sum time.
b) Assume that multiplying two numbers takes 5 units of time, adding two numbers takes 2 units of time, and communicating one number between two processing elements takes 10 units of time. What is the parallel runtime, speedup, and efficiency of your parallel algorithm when run on \( p \) processing elements?

You can assume \( n \) and \( p \) are a power of 2. If you can, write your answer in terms of computation time and communication time components.

**Answer.** First, the sequential runtime is \( 5n + 2(n - 1) = 7n - 2 \). The sequential algorithm does not involve any communication. Now let's write down the execution time equations for the parallel algorithm above.

- Step 1: \( 2 \times 10 \times \frac{n}{p} \) distribute vectors
- Step 2: \( 5n/p \) parallel local multiplication
- Step 3: \( 2 \times \frac{n}{p} - 1 \) parallel local addition
- Step 4: \( (10 + 2) \times \log p \) partial sum and communication

The parallel execution runtime is:

\[
T_p = 20n - 20n/p + 5n/p + 2n/p - 2 + 12\log p = 20n - 13n/p - 2 + 12\log p
\]  

(4)

The speedup is:

\[
S_p = \frac{T_1}{T_p} = \frac{7n - 2}{20n - 13n/p - 2 + 12\log p}
\]  

(5)

The efficiency is:

\[
E_p = \frac{S_p}{p} = \frac{\frac{7n - 2}{20n - 13n/p - 2 + 12\log p}}{p} = \frac{7n - 2}{p \times (20n - 13n/p - 2 + 12\log p)}
\]  

(6)

It is clear that the initial data distribution dominates the execution time. To make the problem not so constrained by this, plus making it a little more interesting, let us assume that the vectors are already located on each processor. In this case, we get the following:

\[
T_p = 5n/p + 2n/p - 2 + 12\log p = 7n/p - 2 + 12\log p
\]  

(7)

\[
S_p = \frac{T_1}{T_p} = \frac{7n - 2}{7n/p - 2 + 12\log p}
\]  

(8)

\[
E_p = \frac{S_p}{p} = \frac{\frac{7n - 2}{7n/p - 2 + 12\log p}}{p} = \frac{7n - 2}{p \times (7n/p - 2 + 12\log p)}
\]  

(9)

c) Calculate the speedup and efficiency assuming that the problem for \( p = 1 \) is that of computing the dot product for two vectors of length 256. Use \( p = 1, 4, 16, 64, \) and \( 256 \), and assume the same time costs as in b).
Answer. The execution time for two vectors of length 256 on one processor is $7 \times 256 - 2 = 1790$.

<table>
<thead>
<tr>
<th>$p$</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>3.8</td>
<td>0.95</td>
</tr>
<tr>
<td>16</td>
<td>11.32</td>
<td>0.7</td>
</tr>
<tr>
<td>64</td>
<td>18.26</td>
<td>0.28</td>
</tr>
<tr>
<td>256</td>
<td>17.72</td>
<td>0.06</td>
</tr>
</tbody>
</table>

d) When executing on 64 processors, how large would $n$ have to be to achieve the same efficiency as achieved on 4 processors for $n = 256$?

Answer. Solving the efficiency equation for $n$ we get

\[
0.95 = \frac{7n - 2}{64} 
\]  
(10)

\[
0.95 \times 64 = \frac{7n - 2}{7n/64 - 2 + 12 \times 6} 
\]  
(11)

\[
60.8 \times (7n/64 - 2 + 72) = 7n - 2 
\]  
(12)

\[
2 - 60.8 \times 2 + 60.8 \times 72 = 7n - 60.8 \times (7n/64) 
\]  
(13)

\[
4258 = (7 - 6.65)n = .35n 
\]  
(14)

\[
4258/.35 = 12166 = n 
\]  
(15)

e) Scaled speedup is defined as the speedup obtained when the problem size is increased linearly with the number of processing elements. That is, if $W$ is chosen as a base problem size for a single processing element, then

\[
\text{Scaled speedup} = \frac{pW}{T_p(pW, p)} 
\]  
(16)

For the dot product problem, calculate the scaled speedups, assuming that the base problem for $p = 1$ is that of computing the dot product for two vectors of length 256. Use $p=1, 4, 16, 64, \text{and} 256$, and assume the same time costs as before.

Answer. We can write the scaled speedup equation as:

\[
SS_p = T_1(pW)/T_p(pW) 
\]  
(17)

Now, we can rewrite our earlier formulae for $T_1$ and $T_p$ to get:

\[
T_1(pW) = 5n \times p + 2(n \times p - 1) = 7np - 2 
\]  
(18)
\[
T_p(pW) = 5n*p/p + 2n*p/p - 2 + 12\log p = 7n - 2 + 12\log p
\]  \hspace{1cm} (19)

The execution time for two vectors of length 256 on one processor is \(7 \times 256 - 2 = 1790\).

<table>
<thead>
<tr>
<th>(p)</th>
<th>Scaled Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>(\frac{7 \times 256 - 2 + 12\log 4}{7 \times 256 - 2 + 12\log 4}) = 3.95</td>
</tr>
<tr>
<td>16</td>
<td>(\frac{7 \times 256 - 2 + 12\log 16}{7 \times 256 - 2 + 12\log 16}) = 15.59</td>
</tr>
<tr>
<td>64</td>
<td>(\frac{7 \times 256 - 2 + 12\log 64}{7 \times 256 - 2 + 12\log 64}) = 61.59</td>
</tr>
<tr>
<td>256</td>
<td>(\frac{7 \times 256 - 2 + 12\log 256}{7 \times 256 - 2 + 12\log 256}) = 243.23</td>
</tr>
</tbody>
</table>