Lecture 26: Board Notes: Parallel Programming Examples

Part A:
Consider the following binary search algorithm (a classic divide and conquer algorithm) that searches for a value X in a sorted N-element array A and returns the index of the matched entry:

```java
BinarySearch(A[0 … N-1], X) {
    low = 0
    high = N-1

    while(low <= high) {
        mid = (low + high) / 2
        if (A[mid] > X)
            high = mid - 1
        else if (A[mid] < X)
            low = mid + 1
        else
            return mid    // we’ve found the value
    }

    return -1      // value is not found
}
```

Question 1:
- Assume that you have Y cores on a multi-core processor to run BinarySearch
- Assuming that Y is much smaller than N, express the speed-up factor you might expect to obtain for values of Y and N.

Answer:
- A binary search actually has very good serial performance and it is difficult to parallelize without modifying the code
- Increasing Y beyond 2 or 3 would have no benefits
- At best we could...
  - On core 1: perform the comparison between low and high
  - On core 2: perform the computation for mid
  - On core 3: perform the comparison for A[mid]
- Without additional restructuring, no speedup would occur
  - ...and communication between cores is not “free”

<table>
<thead>
<tr>
<th>Compare low</th>
<th>Calculate mid</th>
<th>Compare high</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 1</td>
<td>Core 2</td>
<td>Core 1</td>
</tr>
<tr>
<td>Core 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We are always throwing half of the array away!
Question 2:
- Now, assume that $Y$ is equal to $N$
- How would this affect your answer to Question 1?
- If you were tasked with obtaining the best speed-up factor possible, how would you change this code?

Answer:
- This question suggest that the number of cores can be made equal to the number of array elements
- With current code, this will do no good
- Alternative approach is to:
  - Create threads to compare the $N$ elements to the value $X$ and perform these in parallel
  - Then, we can get ideal speed-up ($Y$)
  - Entire comparison can be completed in the amount of time to perform a single computation
- Probably not a great idea to design a processor architecture for just this problem
  - Especially as a binary search should take just $\log_2 N$ operations anyhow

Part B:
(Adapted from https://computing.llnl.gov/tutorials/parallel_comp/)

Note – this example deals with the fact that most problems in parallel computing will involve communication among different tasks

Consider how one might solve a simple heat equation:
- The heat equation describes the temperature change over time given some initial temperature distribution and boundary conditions
- As shown in the picture below, a finite differencing method is employed to solve the heat equation numerically

![Heat Equation Diagram](https://example.com/heat-equation.png)

The serial algorithm would look like:

```python
for iy = 2:(ny - 1)
    for ix = 2:(nx - 1)
        u2(ix, iy) = u1(ix, iy) + cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) + 
                      cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
        U_{x,y+1} = U_{x,y}
                  + c_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{x,y})
                  + c_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})
```

```
Question:
- Assuming we have 4 cores to use on this problem, how would we go about writing parallel code?

Answer:
- We would need to partition and distribute array elements such that they could be processed by different cores
- Given the partitioning shown at right...
  - Interior elements are independent of work being done on other cores
  - Border elements do dependent on working being done on other cores – and we must set up a communication protocol
- Might have a MASTER process that sends information to workers, checks for convergence, and collects results
  - WORKER process calculates solution
Part C:
Consider the following piece of C-code:

```c
for(j=2; j<=1000; j++)
    D[j] = D[j-1] + D[j-2];
```

The assembly code corresponding to the above fragment is as follows:

```
addi r2, r2, 1000
Loop:  lw  r1, -16(rX)
       lw  r2, -8(rX)
       add r3, r1, r2
       sw  r3, 0(rX)
       addi r1, r1, 8
       bne r1, r2, Loop
```

Assume that the above instructions have the following latencies (in CCs)

- addi: 1 CC
- lw:  5 CCs
- add: 3 CCs
- sw:  1 CC
- bne: 3 CCs

**Question 1:**
How many cycles does it take for all instructions in a single iteration of the above loop to execute?

**Answer:**
- The first instruction is executed 1 time
- The loop body is executed 998 times

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Number of times run</th>
<th>Number of cycles</th>
<th>Total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lw</td>
<td>999</td>
<td>5</td>
<td>4995</td>
</tr>
<tr>
<td>lw</td>
<td>999</td>
<td>5</td>
<td>4995</td>
</tr>
<tr>
<td>add</td>
<td>999</td>
<td>3</td>
<td>2997</td>
</tr>
<tr>
<td>sw</td>
<td>999</td>
<td>1</td>
<td>999</td>
</tr>
<tr>
<td>addi</td>
<td>999</td>
<td>1</td>
<td>999</td>
</tr>
<tr>
<td>bne</td>
<td>999</td>
<td>3</td>
<td>2997</td>
</tr>
</tbody>
</table>

This is our baseline … now, let’s see if we can do better.
Question 2:
When an instruction in a later iteration of a loop depends on a value in an earlier iteration of the same loop, we say there is a loop-carried dependence between iterations of the loop.

- Identify the loop-carried dependencies in the above code
- Identify the dependent program variable and assembly-level registers
  - (Ignore the loop counter j)

Answer:
- Array elements D[j] and D[j-1] will have loop carried dependencies
- These affect r3 in the current iteration and r4 in the next iteration

Question 3:
In previous assignments, you used looped unrolling to reduce the execution time of a loop.

- Apply loop unrolling to this loop
- Then, consider running this code on a 2-node distributed memory message-passing system
- Assume that we are going to use message passing and will introduce 2 new operations:
  - send (x,y) – which sends the value y to node x
  - receive (x,y) – which waits for the value being sent to it
- Assume that:
  - send takes 3 cycles to issue
  - receive instructions stall execution on the node where they are executed until they receive a message
    - Once executed, receives also take 3 cycles to process

Given the code below, compute the number of cycles it will take for the loop to run on the message passing system:
- Note: the loop is unrolled 4 times
### Loop running on node 1:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi r2, r0, 992</td>
<td>1</td>
</tr>
<tr>
<td>lw r1, -16(rX)</td>
<td>2, 3, 4, 5, 6</td>
</tr>
<tr>
<td>lw r1, -8(rX)</td>
<td>7, 8, 9, 10, 11</td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>add r3, r2, r1</td>
<td>12, 13, 14</td>
</tr>
<tr>
<td>add r4, r3, r2</td>
<td>15, 16, 17</td>
</tr>
<tr>
<td>send (2, r3)</td>
<td>18, 19, 20</td>
</tr>
<tr>
<td>send (2, r4)</td>
<td>21, 22, 23</td>
</tr>
<tr>
<td>sw r3, 0(rX)</td>
<td>24</td>
</tr>
<tr>
<td>sw r4, 0(rX)</td>
<td>25</td>
</tr>
<tr>
<td>receive (r5)</td>
<td>33, 34, 35</td>
</tr>
<tr>
<td>add r6, r5, r4</td>
<td>36, 37, 38</td>
</tr>
<tr>
<td>add r1, r6, r5</td>
<td>39, 40, 41</td>
</tr>
<tr>
<td>send (2, r6)</td>
<td>42, 43, 44</td>
</tr>
<tr>
<td>send (2, r1)</td>
<td>45, 46, 47</td>
</tr>
<tr>
<td>sw r5, 16(rX)</td>
<td>48</td>
</tr>
<tr>
<td>sw r6, 24(rX)</td>
<td>49</td>
</tr>
<tr>
<td>sw r1, 32(rX)</td>
<td>50</td>
</tr>
<tr>
<td>receive (r2)</td>
<td>57, 58, 59</td>
</tr>
<tr>
<td>sw r2, 40(rX)</td>
<td>60</td>
</tr>
<tr>
<td>addi rX, RX, 48</td>
<td>61</td>
</tr>
<tr>
<td>bne rX, r2, loop</td>
<td>62, 63, 64</td>
</tr>
<tr>
<td>clean up cases outside loops</td>
<td>Assume ~ 20 cycles total</td>
</tr>
</tbody>
</table>

### Loop running on node 2:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi r3, r0, 0</td>
<td>in parallel with first loop</td>
</tr>
<tr>
<td>receive (r7)</td>
<td>21, 22, 23</td>
</tr>
<tr>
<td>receive (r8)</td>
<td>24, 25, 26</td>
</tr>
<tr>
<td>add r9, r8, r7</td>
<td>27, 28, 29</td>
</tr>
<tr>
<td>send (1, r9)</td>
<td>30, 31, 32</td>
</tr>
<tr>
<td>receive (r7)</td>
<td>45, 46, 47</td>
</tr>
<tr>
<td>receive (r8)</td>
<td>48, 49, 50</td>
</tr>
<tr>
<td>add r9, r8, r7</td>
<td>51, 52, 52</td>
</tr>
<tr>
<td>send (1, r9)</td>
<td>54, 55, 56</td>
</tr>
<tr>
<td>receive (r7)</td>
<td>74, 75, 76</td>
</tr>
<tr>
<td>receive (r8)</td>
<td>77, 78, 79</td>
</tr>
<tr>
<td>add r9, r8, r7</td>
<td>80, 81, 82</td>
</tr>
<tr>
<td>send (1, r9)</td>
<td>83, 84, 85</td>
</tr>
<tr>
<td>receive (r7)</td>
<td>104, 105, 106</td>
</tr>
<tr>
<td>receive (r8)</td>
<td>107, 108, 109</td>
</tr>
<tr>
<td>add r9, r8, r7</td>
<td>110, 111, 112</td>
</tr>
<tr>
<td>send (1, r9)</td>
<td>113, 114, 115</td>
</tr>
<tr>
<td>addi r3, r3, 1</td>
<td>116</td>
</tr>
<tr>
<td>bne r3, 984, loop</td>
<td>117, 118, 119</td>
</tr>
</tbody>
</table>
Answer:
Based on the timing information derived above, the time to complete 12 iterations of the loop is:
   - $123 - 12 + 1 = 112$ cycles

Thus, the time to complete the entire loop is:
   - Startup: $11$ cycles
   - Main loop: $\lfloor \frac{1000}{12} \rfloor \times 112 = 83 \times 112 = 9296$ cycles
   - Clean up: $20$ cycles

Thus, the total number of cycles for this code is: $11 + 9296 + 20 = 9327$ cycles

This gives us a speedup of: $\frac{17983}{9327} \approx 1.928$
   - However, speedup does not come entirely from multiple cores
   - Also get benefit from loop unrolling.

Question 4:
In above example, made a slightly idealistic assumption. What is it?
   - Hint: see bold text.

Answer:
   - Any sent message is received instantaneously in the next CC.

Hidden questions:
A. What interconnection latency is tolerable?
   - 12 iterations of the loop involve 12 sends
   - Let’s assume the time to send a message is $N$ cycles

   Need to satisfy:
   $11 + \lfloor (112+12N)^*83 \rfloor + 20 < 17983$

   If we solve for $N$, we see that $N$ is equal to 8.69; thus, practically, $N$ must be less than 9.

B. Not unreasonable to expect $N$ to be on the order of 4 cycles. If so, what is the speedup obtained?

   New number of cycles:
   $= 11 + \lfloor (112 +12*4)^*83 \rfloor + 20$
   $= 11 + 13280 + 20$
   $= 13280$

   Speedup becomes: $\frac{17983}{13280} = 1.35$

Take away: loop with dependencies is practically hard to parallelize…