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:

```
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"

Compare low			Calculate mid			Compare high
Core 1			**Core 2**			**Core 1**
			Compare A[mid]			
			Core 3			

We are always throwing half of the array away!

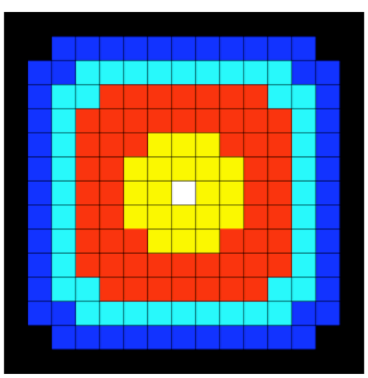
Question 2:
- Now, assume that Y is equal to N
- How would this affect your answer to Question 1?
- If you were tasked with obtaining the best speed-up factor possible, how would you change this code?

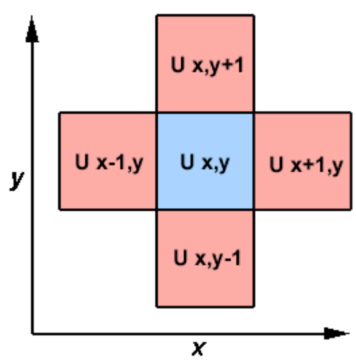
Answer:
- This question suggest that the number of cores can be made equal to the number of array elements
- With current code, this will do no good
- Alternative approach is to:
  - Create threads to compare the N elements to the value X and perform these in parallel
  - Then, we can get ideal speed-up (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





$$U_{x,y} = U_{x,y} + C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy}) + C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$

The serial algorithm would look like:

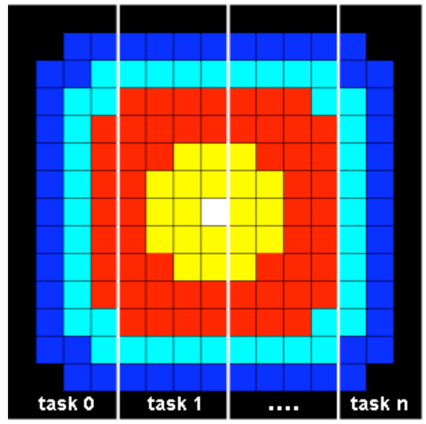
```
for iy = 2:(ny – 1)
        for ix = 2:(nx – 1)
                u2(ix, iy) =   u1(ix, iy) +
                               cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
                               cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
```

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




```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time
    send neighbors my border info
    receive from neighbors their border info

    update my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time

    non-blocking send neighbors my border info
    non-blocking receive neighbors border info

    update interior of my portion of solution array
    wait for non-blocking communication complete
    update border of my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```

Part C:
Consider the following piece of C-code:

```
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

Instruction	**Number of times run**	**Number of cycles**	**Total cycles**
addi	1	1	1
lw	999	5	4995
lw	999	5	4995
add	999	3	2997
sw	999	1	999
addi	999	1	999
bne	999	3	2997
			17983

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:

	addi r2, r0, 992	1	
	lw r1, -16(rX)	2, 3, 4, 5, 6	
	lw r1, -8(rX)	7, 8, 9, 10, 11	
loop	add r3, r2, r1	12, 13, 14	65, 66, 67
	add r4, r3, r2	15, 16, 17	68, 69, 70
	send (2, r3)	**18, 19, 20**	71, 72, 73
	send (2, r4)	21, 22, 23	74, 75, 76
	sw r3, 0(rX)	24	77
	sw r4, 0(rX)	25	78
	receive (r5)	33, 34, 35	86, 87, 88
	add r6, r5, r4	36, 37, 38	89, 90, 91
	add r1, r6, r5	39, 40, 41	92, 93, 94
	send (2, r6)	42, 43, 44	95, 96, 97
	send (2, r1)	45, 46, 47	98, 99, 100
	sw r5, 16(rX)	48	101
	sw r6, 24(rX)	49	102
	sw r1, 32(rX)	50	103
	receive (r2)	57, 58, 59	116, 117, 118
	sw r2, 40(rX)	60	119
	addi rX, RX, 48	61	120
	bne rX, r2, loop	62, 63, 64	121, 122, 123
	clean up cases outside loops	*Assume ~ 20 cycles total*	

Loop running on node 2:

	addi r3, r0, 0	in parallel with first loop	
loop	**receive (r7)**	**21, 22, 23**	
	receive (r8)	24, 25, 26	
	add r9, r8, r7	27, 28, 29	
	send (1, r9)	30, 31, 32	
	receive (r7)	45, 46, 47	
	receive (r8)	48, 49, 50	
	add r9, r8, r7	51, 52, 52	
	send (1, r9)	54, 55, 56	
	receive (r7)	74, 75, 76	
	receive (r8)	77, 78, 79	
	add r9, r8, r7	80, 81, 82	
	send (1, r9)	83, 84, 85	
	receive (r7)	104, 105, 106	
	receive (r8)	107, 108, 109	
	add r9, r8, r7	110, 111, 112	
	send (1, r9)	113, 114, 115	
	addi r3, r3, 1	116	
	bne r3, 984, loop	117, 118, 119	

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:
  - floor(1000 / 12) x 112 = 83 x 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: 17983 / 9327 = ~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 + [(112+12N)*83)] + 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 + [(112 +12*4)*83] + 20
= 11 + 13280 + 20
= 13280

Speedup becomes: 17983 / 13280 = 1.35

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