Lecture 27: 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 ($\log_2(N)$)
- 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"

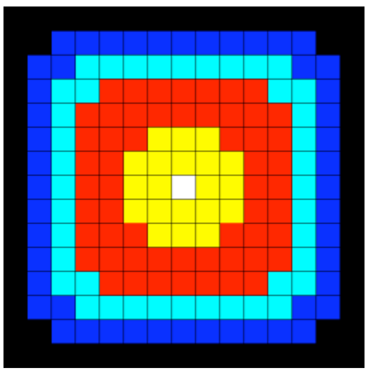
Compare low			Calculate mid			Compare high
Core 1			**Core 2**			**Core 1**
			Compare A[mid]			
			Core 3			

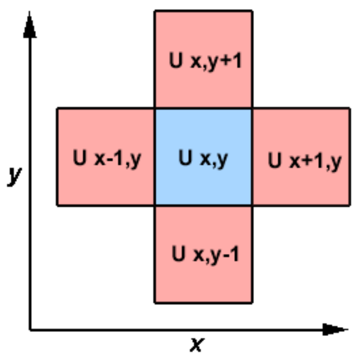
We are always throwing half of the array away!

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 (i.e. approximating derivatives)





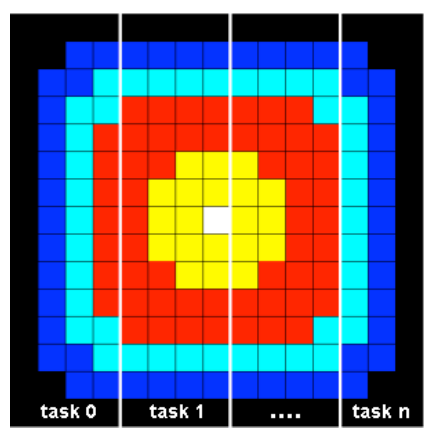
$$U_{x,y} = U_{x,y} + C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy}) + C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$

Question: **What would the serial algorithm look like?**

```
for (i=2; i < (y – 1); i++) {
        for (j=2; j < (x – 1); j++) {
                u[x,y] =        u[x,y] +
                                cx * (u[j+1, i] + u[j-1, i] - 2*u[j,i]) +
                                cy * (u[j, i+1] + u[j, i-1] – 2*u[j,i])
        }
}
```

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
```

Parallelized code

```
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
```

Slightly more efficient code
(blocking will become more relevant when material in next lecture packet is discussed)

Part C:

Consider the following piece of C-code:

```
for(j=2; j<=1001; j++)
      D[j] = D[j-1] + D[j-2];
```

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

```
        addi   r10, r10, 4004
        addi   rx, rx, 8
Loop:   lw     r1, -8(rX)
        lw     r2, -4(rX)
        add    r3, r1, r2
        sw     r3, 0(rX)
        addi   rx, rx, 8
        bne    rx, r10, Loop
```

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

addi:	4 CC	sw:	4 CC
lw:	5 CCs	bne	3 CCs

add: 4 CCs

Question 1:
How many cycles does it take for all instructions in a single iteration of the above loop to execute? (Assume pipelined and non-pipelined datapaths with perfect branch prediction…)

Answer – non pipelined…

- The first 2 instructions are executed 1 time
- The loop body is executed 1000 times

Instruction	Number of times run	Number of cycles	Total cycles
addi	1	4	4
addi	1	4	4
lw	1000	5	5,000
lw	1000	5	5,000
add	1000	4	4,000
sw	1000	4	4,000
addi	1000	4	4,000
bne	1000	3	3,000
			25,008

Answer – pipelined…

- We can apply the formula: NT + (S-1)T – as there are no dependencies that should cause stalls
  - (2+1000x6)(T) + (5-1)(T)
  - (6002)(T) + 4(T)
  - 6006T
- In other words, **6006 cycles**

This is our baseline … now, let's see if we can do better
- Note that the pipelined version is

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:
How can we parallelize / improve the performance of this code?

Answer:

- Hard to parallelize with loop carried dependence…
- Best approach is to unroll loop

Let's re-write our C-code…

```
for(j=2; j<=1005; j+=5) {
        D[j]        = D[j-1] + D[j-2];
        D[j+1]      = D[j] + D[j-1];
        D[j+2]      = D[j+1] + D[j];
        D[j+3]      = D[j+2] + D[j+1];
        D[j+4]      = D[j+3] + D[j+2];
}
```

```
        addi    r10, r10, 4004
        addi    rx, rx, 8

Loop:   lw      r1, -8(rX)          # load d(j-2)
        lw      r2, -4(rX)          # load d(j-1)

        add     r3, r1, r2          # calculate d(j)
        sw      r3, 0(rX)           # store d(j)

        add     r4, r2, r3          # calculate d(j+1)
        sw      r4, 0(rX)           # store d(j)

        add     r5, r3, r4          # calculate d(j+1)
        sw      r5, 0(rX)           # store d(j)

        add     r6, r4, r5          # calculate d(j+1)
        sw      r6, 0(rX)           # store d(j)

        add     r7, r5, r6          # calculate d(j+1)
        sw      r7, 0(rX)           # store d(j)

        addi    rx, rx, 24          # update counter
        bne     rx, r10, Loop
```

Now we can calculate new times…

For the multi-cycle version…

Instruction	Number of times run	Number of cycles	Total cycles
addi	1	4	4
addi	1	4	4
lw	200	5	1000
lw	200	5	1000
add	200	4	800
sw	200	4	800
add	200	4	800
sw	200	4	800
add	200	4	800
sw	200	4	800
add	200	4	800
sw	200	4	800
add	200	4	800
sw	200	4	800
addi	200	4	800
bne	200	3	600
			11,412

For the pipelined version:

- We can apply the formula: NT + (S-1)T – as there are no dependencies that should cause stalls
    - (2+200x14)(T) + (5-1)(T)
    - (6002)(T) + 4(T)
    - 6006T
- In other words, **2806 cycles**

Comparing un-rolled to non-unrolled…

Multi-cycle: 25,008 vs. 11,412: unrolled is 2.19X faster
Pipelined: 6,006 vs. 2,806: unrolled is 2.14X faster