Max Shi  
CS 383  
Professor Barbalace  
December 5, 2019  
I pledge my honor that I have abided by the Stevens Honor System.

Homework 5

6.3.1.

There is minimal speedup for the binary search, without modifying the code. Because we are partitioning the array and only using one proportion out of each partition, there is no work to be parallelized, except possibly the comparison of the low and high numbers, as well as the computation and comparison of mid. This is 3 separate computations, thus no speedup can be gained after 3 parallel processors.

6.3.2.

Making the number of cores equal to the number of values does not change the speedup of binary search, but I would use an entirely different algorithm to do the search, that is, allowing each thread to make one comparison with the target element to find it in the array. Thus, this algorithm would finish in a single operation, giving ideal speedup, as Y/1 = Y times speedup, and because Y = N, this is ideal.

6.9.1.

a.

|  |  |
| --- | --- |
| Core 1 | Core 2 |
| A3, A2 | B1, B4 |
| A1, A4 | B1, B4 |
| A1 | B2 |
| A1 | B3 |

It will take 4 cycles to execute, with four issue slots wasted due to hazards.

b.

Having two SS processors does not impact it, as only one thread can run on one core at a time. Thus, with the two cores on CPU 1, the solution is the same:

|  |  |
| --- | --- |
| CPU 1 |  |
| Core 1 | Core 2 |
| A3, A2 | B1, B4 |
| A1, A4 | B1, B4 |
| A1 | B2 |
| A1 | B3 |

Again, it takes 4 cycles, with four issue slots wasted.

c.

|  |  |
| --- | --- |
| FU 1 | FU 2 |
| A1 | A2 |
| A1 | B1 |
| A1 | B1 |
| A3 |  |
| A4 |  |
| B2 | B3 |
| B4 |  |
| B4 |  |

8 cycles to complete, 4 issue slots wasted due to hazards.

d.

|  |  |
| --- | --- |
| FU1 | FU2 |
| A1 | B1 |
| A1 | B1 |
| A1 | B2 |
| A2 | B3 |
| A3 | B4 |
| A4 | B4 |

7 cycles to complete, no issues slots wasted due to hazards.

6.11.1

MIMD is similar to SISD, just splits the data. Thus, LEGv8 code is simple:

mov x10, #0, =lower\_bound\_i //starting value of i  
ldr x11, =upper\_bound\_i //upper bound of i  
mov x12, #0 //set initial j  
mov x13, #3000 //set upper j  
ldr x14, =xarray //initial pointer of x array  
ldr x15, =yarray  
.loop  
cmp x10, x11 //compare i  
bge exit //exit if loop is done  
lsl x0, x10, #3 //offset for i  
ldr x1, [x14, x0] //load &xarray[i]  
mov x12, #0 //set j to 0  
.innerloop  
cmp x12, x13 //compare j  
bge endinner //end if inner loop is over  
lsl x2, x12, #3 //offset for j  
ldr x3, [x15, x2] //load yarray[j]  
ldr x3, [x3, x0] //load yarray[j][i]  
add x3, x3, #200 //add 200  
str x3, [x1, x2] //store xarray[i][j]  
add x12, x12, #1 //increment j  
bl innerloop  
.endinner  
add x10, x10, #1 //increment j  
bl loop //loop back  
.exit  
br x30  
  
To split this among four processors, I would change the lower and upper bounds of y so that each processor gets a 500 interval, i.e. 0 to 500, 500 to 1000, etc. Thus, it splits the work in 4, and gains a 4x speedup.

6.11.2.

Let the address of xarray be stored in x14 and the address of yarray be stored in x15. Let V be vector registers, and let LDURDV load a vector (a row of the matrix) and let STURDV store a vector. Also, let FADDD add a constant to each element in a vector. Let LDURDVS load a subvector, as in, a vector of all elements with the specified offset, and let STURDVS do the same thing, but store it.

ldr x9, =lower\_bound\_j  
ldr x10, =upper\_bound\_j  
ldr x11, =x\_array //x array pointer  
ldr x12, =y\_array //y array pointer  
.loop  
cmp x9, x10 //compare i   
bge end   
lsl x0, x9, #3 //offset for i  
LDURDVS V1, [x12, x0] //load yarray[][i] (all elements in y with second index i)  
FADDD V1, V1, #200 //add 200 to all elements  
STURDVS V2, [x11, x0] //store the new vector in all elements in xarray[i]  
add x9, x9, #1 //increment counter  
bl loop //loop again  
.end  
br x30 //exit

This code is much shorter, and only requires keeping a pointer to i, as j is handled by the vector properties of the processor. Also, there does not have to be an inner loop, as we no longer iterate over j. Thus, I have reduced the number of instructions from 27 to 15. Because this is an eight way processor, we can split the workload up into portions of 125, i.e. 0 to 125, 125 to 250, etc, resulting in an 8x speedup.

6.19.1.

I would partition the vector CPU 3 to do subtraction of Y from X (X-Y) in batches of 8 entries of the matrices, and store those in memory. Then, I would concurrently use the FP CPU 1 to load those values from memory as they become available to do floating point comparisons on whether or not those values are 0 or positive. Greater than or equal to 0 would signify that X was greater than Y. Then, we can simply use CPU 1 to count the number of times this occurs, and record this number. We would not actually need to use CPU 2.

6.19.2.

Adding a floating point comparison instruction that allows us to compare greater than or equal to between two vectors would speed up the operation, and allow us to load the vector result of this operation into the faster CPU 2 to do integer comparisons on 1 and 0 in order to count the number of times X is greater than Y.

Non-textbook.

1. Ring configuration
   1. Total bandwidth – 16 Gbit/s
   2. Minimum Bisection Bandwidth – 2 Gbit/s
   3. 1 link can fail, which would result in taking the opposite way around the ring. Two links failing could disconnect a node.
2. Hypercube
   1. Total bandwidth – number of edges = 32 Gbit/s
   2. Minimum Bisection Bandwidth = 8 Gbit/s
   3. 3 links can fail before the corner is cut off from the rest of the cube, so 3 is the answer.
3. 2d mesh
   1. Total bandwidth = 24 Gbit/s
   2. Minimum Bisection Bandwidth = 4 Gbit/s
   3. 1 link can fail before the corner is cut off, so 1 link.
4. Fully connected network
   1. Total bandwidth = 120 Gbit/s
   2. Minimum Bisection Bandwidth = 64 Gbit/s
   3. For a node to be completely disconnected, it has to fail all 15 connections it has. Thus, 14 connections can fail and we can still ensure all nodes are connected.