CSC/ECE 506: Architecture of Parallel Computers
Sample Test 1 (refactored)

This was a 120-minute open-book test. You were allowed use the textbook and any course notes that you had. You were not allowed to use electronic devices. You were to answer five of the six questions. Each question was worth 20 points. If you answered all six questions, your five highest scores counted.

Question 1. *[Average score 16.8] (1 point each)* For each of the following terms or concepts, choose the programming model with which it is most closely associated:

- Data parallel (write "D")
- Message-passing ("M")
- Shared address space ("S")

(a) Array processor	*Ans.:* D	(k) Local array indexes	*Ans.:* M
(b) Buffer-management overhead	*Ans.:* M	(l) Lock operations	*Ans.:* S
(c) Coherence problems	*Ans.:* S	(m) Logically single thread of control	*Ans.:* D
(d) Peterson's algorithm	*Ans.:* S	(n) Loosely coupled multiprocessor	*Ans.:* M
(e) Distributed-memory multiprocessor	*Ans.:* M	(o) NUMA machine	*Ans.:* S
(f) GPU	*Ans.:* D	(p) **receive** operation	*Ans.:* M
(g) Barrier synchronization	*Ans.:* S	(q) CUDA Kernel Method	*Ans.:* D
(h) **for_all** loop	*Ans.:* D	(r) SPMD	*Ans.:* D
(i) cluster architecture	*Ans.:* M	(s) Shared-memory multiprocessor	*Ans.:* S
(j) Thread blocks	*Ans.:* D	(t) Tightly coupled multiprocessor	*Ans.:* S

The most frequently missed part was (k). More people thought local array indexes are associated with data-parallel algorithms. In message-passing, when an array is divided up among the different processors, each processor uses a local index to iterate through its portion of the array. On data-parallel machines, registers aren't used in this fashion because the different processing elements go from one element to another of their array portion as dictated by the control processor. However, data-parallel machines do have local index registers, so I gave credit for the "D" answer.

The next most frequently missed part was (o). A NUMA machine *can* address all its memory, so it can run shared-memory programs. Thus, the shared-memory model is most closely associated with it.

Question 2. <GPU L3-video & Program 1> (George)
(a) *(2 points per blank, max. 10 points)* Modern GPUs can support a maximum of 1024 blocks and maximum of 65536 threads per block. Fill in the blanks in the host code given below to check for these two conditions before invoking the kernel. *Note:* This is not the complete code. Memory allocation and error-checking have been removed so the code can fit on one page.

```
#include <stdio.h>
#include <cuda_runtime.h>
#define MAX_THREADS __65536__
#define MAX_BLOCKS ___1024__
```

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
         C[i] = A[i] + B[i];
}

int main(int argc, char * argv[]){
  ....
 // here numElements = argv[1]

 int threadsPerBlock = 256;
  int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;

 // Check the boundary conditions for max number of elements that GPU can handle

  if(numElements> __MAX_THREADS__ * _MAX_BLOCKS_ ) {
    printf("The given number of elements cannot be executed in one go; tiling   is
required\n");
    exit(-1);
  }

  .......

 // The idea here is to meet the physical limitations of the GPUs before calling the kernel

  while( _blocksPerGrid_ > MAX_BLOCKS ) {

    int _threadsPerBlock_  *= 2;
    int blocksPerGrid = (int)((numElements+threadsPerBlock-1)/threadsPerBlock);
  }

 // Launch the Vector Add CUDA Kernel

  printf("CUDA kernel launch with %d blocks and %d threads per block\n",
       blocksPerGrid, threadsPerBlock);

  vectorAdd <<< blocksPerGrid, threadsPerBlock >>>(d_A, d_B, d_C, numElements);
  return 0;
}
```

(b) *(2 points)* What will be the output of the program if the executable file is executed like this: "`./vectorAdd 68400000`"?

Answer: The program will hit the return condition and print, "The given number of elements cannot be executed in one go; tiling is required."

(c) *(5 points)* Fill in the blanks in the output of the printf before the kernel invocation when the file is executed like this : "`./vectorAdd 307969`"

CUDA kernel launch with _602_ blocks and _512_ threads per block

Initially the block number will be calculated as:
blocksPerGrid = (307969+ 256 -1)/256 = 1204
Now 1204> 1024. Hence the while loop below will execute.
While loop:
threadsPerBlock = 2 × 256 = 512
blocksPerGrid = (307969 + 512 – 1)/512 = 602.5 → 602

Since 602 is less than 1024, the loop will execute only once.

(d) *(3 points)* The **if** statement is included to turn off threads that are not operating on grid elements. In part (c), how many threads does the if statement "turn off"?

Answer: total number of threads created = 602 × 512 = 308,224

The number of threads which are not "turned off" = 307,969

Hence the number of threads which will be turned off = 308,224– 307,969 = **255**

Question 3. <Amdahl's law. L3>
Suppose you're tasked with working with a partially parallelizable program (there is a portion of the program that *must* be executed serially). When executed with 1 processor, the execution time (T_0) is 10 seconds. When executed with N = 3 processors, the execution time (T_1) is 8 seconds.

(a) *(3 points)* What is the speedup achieved by the second execution over the first?

Answer: $S = \frac{T_0}{T_1} = \frac{10}{8} = 1.25$

(b) *(5 points)* What fraction of execution time ***s*** consists of serial code?

Answer:

$$\frac{T_1}{T_0} = s + \frac{(1-s)}{p}$$
$$s = \frac{p\,T_1/T_0 - 1}{p-1} = \frac{3\times 8/10 - 1}{3-2} = 0.7$$

(c) *(5 points)* How long will it take the program to execute when run with 4 processors?

Answer:

$$T_2 = s \times T_0 + \frac{1-s}{p} \times T_0 = 0.7 \times 10 + \frac{0.3}{4} \times 10 = 7.75$$

(d) *(3 points)* What is the limit on achievable speedup for this algorithm?

Answer:

$$max\ speedup = \frac{1}{s} = 10/7 \approx 1.4286$$

(e) *(5 points)* According to Gustafson's law, when more processors are available, workloads will expand to take advantage of them. An expanded version of the program in this question can be executed in 8 seconds, but it requires 100 processors. What is the amount of execution time if only 1 processor is used?

You can assume the fraction of execution time ***s*** consisting of serial code stays the same.
Answer:

$$8 = s \times T'_0 + \frac{1-s}{p} \times T'_0 = 0.7 \times T'_0 + \frac{0.3}{100} \times T'_0$$
$$T'_0 = 8/(0.7 + 0.003) = \frac{800}{703} \approx 11.3798$$

Question 4. (a) *(18 points)* Consider the following algorithm. If we are to parallelize this algorithm for each **for** loop, fill in the table appropriately for each variable used.

```
for (i = 0; i < n; i++){
      for (j = 0; j < n; j++){
            for (k = 0; k < n; k++){
                  if (d[i][j] - d[j][k] < d[i][k])     {
                        b[j][k] = k*j;
                        d[i][j] = d[j][k] + d[i][k];
                  }
```

```
                }
        }
}
```

Which loop parallelized? →	**for** *i*	**for** *j*	**for** *k*
Read-only	*n*	*i*, *n*	*i*, *j*, *n*
R/W non-conflicting		*b*	*b*
R/W conflicting	*d*, *i* ,*j*, *k*, *b*	*d*, *j*, *k*	*d*, *k*
Private	*i*, *j*, *k*	*j*, *k*	*k*
Shared	*d*, *n*, *b*	*d*, *i*, *n*, *b*	*d*, *i*, *j*, *n*, *b*

In the **for** *i* loop, *d* is R/W conflicting, because the `i` subscript appears in the second position on the left-hand side of the assignment statement and in the first position on the right-hand side. This means that, for example, the `i=1` task will write `d[2][1]`, while the `i=2` task will read it.

In the **for** *j* loop, *d* is R/W conflicting. Let's say that `i=1`. Then all of the the **for** *j* iterations use the value of 1 for `i`, regardless of what value they are using for `j`.. For example, the `j=1` task will write `d[1][1]`, but the `j=2` task will also read `d[1][1]`, when it reads `d[i][k]` on the 1st iteration of the inner **for** *k* loop. In fact, all the **for** *j* tasks will read `d[1][1]`, which is written by the `j=1` task.

In the **for** *k* loop, *d* is R/W conflicting. Let's say that `i=1` and `j=2`, and the **for** *k* iterations for these values of *i* and *j* are running in parallel. Then all tasks are writing `d[2][1]`, and `d[2][1]` is also read by the `k=1` task.

(b) *(2 points)* Do any of the shared variables need to be protected by a critical section? Explain.

Answer: In the **for** *i* parallelization, *d* and *b* need to be protected by a critical section, because multiple processes can update the same elements of this array. Similarly for the *j* and *k* loops, *d* needs to be protected by a critical section.

Question 5. This question concerns the communication requirements for the message-passing version of the Ocean application. We assume that `n` = 128 and `nprocs` = 16.

(a) *(2 points)* When a row is sent or received, how many values are transferred?

Answer: 128. The 0th and (*n*+1)st elements of the row do not participate in computations for the row above (or row below), so they do not need to be sent.

(b) *(2 points)* How many floating-point values need to be sent by a "boundary" processor (processors 0 and 15) during a single iteration of the **while** loop that extends from line 15 to line 26?

Answer: 128 values, since these processors send only a single row.

(c) *(2 points)* How many floating-point values need to be sent by a non-boundary processor (processors 1 through 14) in a single iteration of the **while** loop?

Answer: 256 values, since these processors send two rows.

(d) *(4 points)* How many floating-point values need to be sent by all processors put together in a single iteration of that **while** loop?

Answer: 14 × 256 + 2 × 128 = 15× 256 = 3,840

(e) *(5 points)* Repeat part (d) for a cyclic assignment of rows. Note that each processor still needs to handle four rows.

Answer: Since there are 16 processors and 128 rows, each processor still needs to handle 8 rows, but for each of those rows, it will need to send 256 elements. That is, except for the two boundary rows, where only 128 elements will be sent. So the number of values sent is

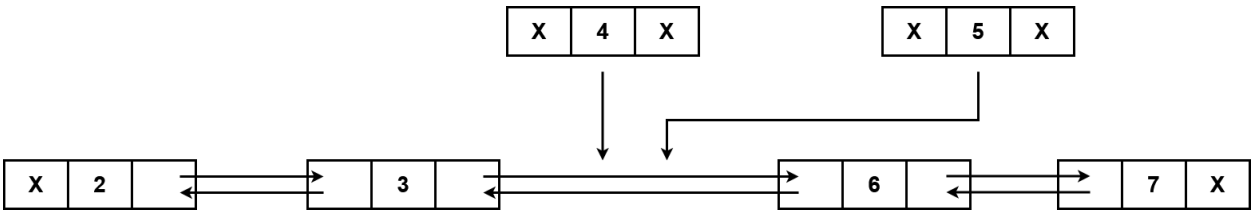
126 × 256+ 2 × 128= 127× 256= 32,512

(f) *(5 points)* Repeat parts (b) through (d), but assuming a 2D partitioning into square blocks.

Answer: Since there are 16 processors, the grid is partitioned into 4 blocks in the horizontal and eight blocks in the vertical dimensions. Each block is a 32 × 32 subgrid. However, there are now four edges where elements need to be sent (unless the edge corresponds to a boundary of the grid itself). So each processor sends 32 × 4 = 128 values, except for the 16 edges (4 on each side) that correspond to boundaries of the grid. Overall, there are 512 (128 × 4) boundary elements in the entire grid.

So the total number of floating-point values sent is 32 × 128 – 512= 4096– 512= 3584 floating-point values.

Question 6. Consider the doubly-linked list below. "X" signifies a null link.



(a) It is desired to insert nodes 4 and 5 between node 3 and node 6. If these insertions take place while a search operation for key = 7 happens in parallel, is there a risk of a non-serializable outcome? If so, explain the problem and how to resolve it.

Answer: Yes, there is a chance of a non-serializable outcome. When two insertions occur at the same node, there will be conflicts due to parallel writes at the same location. The conflicts can be avoided by using a parallelization strategy that uses locks.

The search operation will be successful whether Thread 1 executes first or Thread 2 executes because it will not have conflicts as the insertion and search operations happen at different nodes. If the operations happen at a particular node then there is a chance of conflicts if not synchronized properly whereas if the operations are done at different nodes then there will be no conflicts. [You should run through one possible conflict. I think there is too much hand-waving here.]

(b) In order to prevent conflicts between an insertion and a search, what are the changes that should be made to the code on p. 119 of the 2016 Solihin text to implement the re-traversal for the global-lock approach? Fill in the blanks below to re-implement lines 30 to 38 for doubly-linked lists.

```
setLock(global, WRITE)
if (prev-> deleted || p->deleted || prev->next !=p )
{
   success=0;
 }
 else
 {
 //Insert code below
 newNode -> next = p;
```

```
p->prev = newNode;
newNode -> prev = p->prev;
prev -> next = newNode;
}
 unsetLock(global);
```

(c) Fill in the table below with the number of write locks and read locks that are required for the 1 search operation (key=7) and the given two insertions for all parallelization strategies, viz.,parallelization among readers, global-lock approach and fine-grain lock approach.
Answers below are underlined.

Type	# read locks performed	# write locks performed
Parallelization among readers	1	2
Global-lock approach	1	2
Fine-grain lock approach	1	4

For parallelization among readers, only one lock per operation is required, i.e., 2 write locks for the two insertions and 1 read lock for the search operation. For the global-lock approach, traversals are parallel whereas the execution is sequential. Each operation needs one global read or write lock. So, 2 global write locks are used for the two insertions and 1 global read lock is used for the search operation.
For the fine-grain approach, the insertions lock the cells 3, 4, 6 and 3, 5, 6, respectively. So, for insertions, we need 4 locks and for a search operation, we need a single read lock on cell 7.