CSCI-UA.0480-003

Parallel Computing

Final Exam

Spring 2017 - May 15th (90 minutes)

NAME: **ID:**

- This exam contains 9 questions with a total of 50 points in **four pages.**
- If you have to make assumptions to continue solving a problem, state your assumptions clearly.

1. [2 points] Having shared memory makes it easy for threads to communicate. Why do we have distributed memory machines then?

Because a memory does not scale to serve as many threads/processes as we may want. Moreover, as the number of processes increases, the contention on using any shared resources increases.

2. [3 points] If we have 3 MPI processes, what is the minimum number of *threads* that exist? Can we have more than this minimum? Justify.

The minimum is 3 threads (one thread per process if each process is sequential). We can have more than that if the process itself is multi-threaded (with OpenMP for example).

3. [3 points] When is loop unrolling beneficial in *OpenMP*?
Any one of the reasons will earn the 3 points.

- When threads are too small to amortize the overhead of creating and managing threads.
- When there are not enough cores

4. [6 points] State three different methods in OpenMP that make each thread do a different task (i.e. executing the same loop body but on different data is NOT considered different task).

- If-else or switch-case dependent on thread ID
- Tasks
- Section

5. Suppose we have three processes. Each process has an array p of integers of 3 elements as follows. Process 0 has [1, 2, 4], process 1 has [2. 1, 2], and process 2 has [4, 4, 1]. Suppose all the processes execute the following:

MPI_Reduce(p, q, 3, MPI_INT, MPI_BOR, 1, MPI_COMM_WORLD);

Where p is a pointer to the array and q is a pointer to a receiving array (i.e. another array of 3 integers of values [0, 0, 0]).

a. [8 points] After executing the above function, what will be the content of q for:

- process 0: [0, 0, 0]
- process 1: [7, 7, 7]
- process 2: [0, 0, 0]

b. [1 point] What will happen if one of the processes does not execute the above function?

Deadlock

6. [6 points] State three reasons why a GPU version of a code can be slower than a sequential code even though the code has data parallelism.

• Communication overhead between device memory and host memory

• Blocks require a lot of resources such that they cannot share the SM simulataneously.

• A lot of global memory access

7. Suppose we have the following part of a CUDA kernel that will be executed by two threads: T1 and T2. Assume we have only those two threads in this problem.

```
__global__  int x;
__shared__  int y; // initialized to zero
… //some code not important for this problem
int z;
if (tid == T2)
    { y = 1; }
if (y == 1)
    { z = 1; }
else { z = 2; }
__syncthreads();
if (tid == T1) x = z;
```

What will be the final value (or possible values if there is more than one) of x if:

a. [2 points] T1 and T2 are in the same warp.

1

b. [2 points] T1 and T2 are in the same block but not in the same warp.

There is a race condition here so x can be 1 or 2.

c. [2 points] T1 and T2 are in two different blocks.

2

d. [8 points] Fill in the table with the number of copies of x, y, and z in each scenario.

Scenario: T1 & T2	copies of x	copies of y	copies of z
same warp	1	1	2
same block but not warp	1	1	2
different blocks	1	2	2

8. [2 points] How do we synchronize all threads in a CUDA kernel (i.e. not only threads in the same block but also in different blocks)?

You just need to end the kernel and start a new kernel at the point of synchronization.

9. [5 points] Identify the dependencies loop-carried existent in the following code block, and write a parallel version of the code in OpenMP with the dependencies removed .

```
for (i = 0; i < N - 2; i++) {
        a[i] += a[i + 2] + 5;
        x += a[i];
}
```

First, you need to look at what the sequential version does. It uses *old* versions of A[] elements to generate new versions. So, $a[i]_{new} = a[i+2]_{old} + 5$
So, all what you need to do is:

```
#pragma omp parallel for reduction (+:x)
for (i = 0; i < N - 2; i++) {
        int temp = a[i+2];
        #pragma omp barrier
        a[i] += temp + 5;
        x += a[i];
}
```

We need the barrier to ensure that temp will not get the new version of a[i+2] in case the other thread moves faster.
This is just one way of doing it but there are other implementations too.