CSCI-UA.0480-051: Parallel Computing
Final Exam (Dec 18th, 2020)

Important Notes- READ BEFORE SOLVING THE EXAM

- **If you perceive any ambiguity in any of the questions, state your assumptions clearly and solve the problem based on your assumptions. We will grade both your solutions and your assumptions.**
- **This exam is take-home. It is open-book/notes and you can consult the lectures online. But you cannot communicate with other students or seek answers online.**
- **The exam is posted, on NYU classes, on Dec 18th at 2pm EST.**
- **You must upload one pdf file. The name of that file is your netID.pdf**
- **You have up to 23 hours and 55 minutes to submit on NYU classes.**
- **You are allowed only one submission, unlike assignments and labs.**
- **Your answers must be very focused. You may be penalized for wrong answers and for putting irrelevant information in your answers.**
- **Your answer sheet must have a cover page (as indicated below) and one problem answer per page.**
- **The very first page of your answer must only contain:**
  - **You Last Name**
  - **Your First Name**
  - **Your NetID**
  - **Copy and paste the honor code showed in the rectangle at the bottom of this page.**

Honor code (copy and paste to the first page of your exam)

- You may use the textbook, slides, and any notes you have. But you may not use the internet except to consult the lectures on NYU classes or the course website.
- You may NOT use communication tools to collaborate with other humans. This includes but is not limited to G-Chat, Messenger, E-mail, etc.
- Do not try to search for answers on the internet it will show in your answer and you will earn an immediate grade of 0.
- Anyone found sharing answers or communicating with another student during the exam period will earn an immediate grade of 0.
- **"I understand the ground rules and agree to abide by them. I will not share answers or assist another student during this exam, nor will I seek assistance from another student or attempt to view their answers."**

Problem 1
Suppose we have the following code snippet is running on an eight core processor:

```
1. #pragma omp parallel for
2. for(int i = 0; i <N; i++){
3.     for(int j = i; j < N; j++){
4.         array[i*N + j] = sin(i) + cos(j);
5.     }
6. }
```

(a) [5 points] Given the code above, which **schedule** will be better in terms of performance: static? Or Dynamic? And Why?

[2] The schedule can be dynamic. [If you pick static then you get 1 point instead of 2 because dynamic is more adjusting to side effects like cache misses, etc.]
[3] Because, as loop-index i increases, the number of iterations in inner-loop decreases. Therefore, the work done in each iteration of the outer-loop (parallelized by the pragma) decreases linearly.

(b) [5 points] If we substitute line 1 with **#pragma omp for** will we get same? Better? or worse performance than the original code? Justify your answer.

[2] We will get worse performance.
[3] Without the keyword **parallel** no threads will be created and the code will be executed sequentially.

(c) [5 points] If we move the pragma statement from the outer loop, line 1, to the inner loop, before line 3, will that give better performance? Justify your answer.

[2] We will get worse performance.
[3] You need to mention at least one of the following reasons to get full credit.

- As i increases, we get fewer and fewer inner loop iterations, limiting the parallelism that can be exploited.
- Also, creating and destroying threads in each outer-loop iteration incurs more overhead.

(d) [20 points, 5 for each bullet] Assume *array[]* is an array of int, cache block is 64 bytes, and N is 4096. For each statement below, to be used in line 1 above, indicate whether there may be false sharing and why. Assume we have one thread per outer loop iteration.

Before we solve this problem, let's visualize the access pattern. array[] can be seen as NxN array of int where N = 4096. Based on the code above, each thread will be responsible for 4096/8 = 512 rows. Each row needs 4096/64 = 64 cache blocks. So, there is no possibility of false sharing no matter the schedule.

- #pragma omp parallel for schedule(static, 1)
- #pragma omp parallel for schedule(static, 8)
- #pragma omp parallel for schedule (static, 16)
- #pragma omp parallel for schedule (dynamic)

Problem 2

Suppose we want to write a program to ask a user to enter a positive number N. After that, the program will ask the user to enter N integers. Then, for each one of the N numbers, the program calculates the factorial of that number and prints the results on the screen. For example, if the user enters N = 3 and then enters: 3, 7, and 9, the program must print on the screen the result of: 3!, 7!, and 9!.

We want to write that problem in OpenMP such that we have one thread responsible for each factorial. So, for the example above, we need three threads, one to calculate 3!, one for 7!, and one for 9!.

(a) [5 points] If you must choose between using sections and tasks, which technique will you use? And Why? Do not write code.

[2] I will use tasks.

[3] Because, to use sections, you need to know, at programming time, how many parts will be executed in parallel. And, since N is known only at execution time, so tasks is the way to go.

(b) [5 points] If we decide to use a for-loop over the N numbers and each iteration calculates the factorial of one of the numbers, what will be the best schedule? And why?

[2] The best schedule will be dynamic or auto (any of the two will get you the full credit for that part. If you pick static, you get 1 point only).

[3] It is obvious that each thread will have different amount of work because the the computations needed for factorial(x) depends on x.

(c) [5 points] Suppose that the N numbers the user entered are stored in an array. We want to create N threads. Each thread must go over each one of the N numbers and add to it 2^x where x is the thread ID. So, thread with ID 0 will add 1, thread with ID 1 will add 2, thread with ID 2 will add 4, etc. If the user entered 3, 7, and 9, as above, then the first number will be updated by the three threads to be $3 + 2^0 + 2^1 + 2^2$, the second number will be updated to be $7 + 2^0 + 2^1 + 2^2$, etc. Now, as you can see, there is a race condition at each number of the N numbers (when more than one thread try to update the number). How will you deal with that issue? You have four choices: atomic, critical, critical with name, and locks. Which one will you pick and why?

[2] I will use locks.

[3] Each number update by a thread is a critical section.

The form 2^x is not one of the allowed forms in atomic.

Critical will prohibit all threads from updating any other number if one thread is updating one element. So, we must give a name for each critical section. But, to do so, we must know how many critical sections we have at programming time. Since number of critical sections = N, which is entered by the user, then we don't have this information.

Problem 3
Answer the following questions about GPUs and CUDA.

(a) [5 points] Suppose we have a GPU with 8 SMs. If we launch a kernel with 16 blocks, will that generate an error because the number of blocks is larger than the number of SMs? If yes, state why. If no, state what will happen.

[2] This will not generate any errors.
[3] The GPU will assign the first 8 blocks to the 8 SMs, one block per SM. Then it will see it the SMs have enough resources to take another block. If so, the other blocks will be assigned too. If not, they will be kept in a ready queue in the GPU till a block finishes and another SM becomes available.

(b) [5 points] If we have a GPU that has 64 SPs per SM. Can we launch a block with 128 threads? If no, state why. If yes, state how an SM with 64 SPs will deal with a block of 128 threads.

[2] Yes, we can do that.
[3] A block, once assigned to an SM, is divided into warps. Each warp is 32 threads. If the number of threads in the block is not multiple of 32, the last warp will have less threads. The number of SPs inside SM will have at least 32 SPs. In high-end GPUs they have a number of SPs that is multiple of 32. So, warps will be scheduled for execution. Then, in context switch, another warp from the block, etc, till the whole block is done executing.

(c) [4 points] When knowing about the concept of warps you can write better CUDA programs. State two reasons for why knowing about warps can enhance your CUDA coding.

- You will pick number of threads per block to be multiple of 32 to ensure better utilization of SPs.
- You will pay attention to the if-else in your kernel to ensure no branch divergence.

(d) [6 points] State three characteristics of a problem that makes it a good candidate for GPU instead of multicore.
[Any three from the ones below will give you full credit. Each item is 2 points]

- Computational intensive
- Independent computations
- Similar computations
- Problem size is big enough to ensure enough parallelism.

Problem 4
Some interesting questions about MPI

(a) [6 points] Suppose, in MPI_COMM_WORLD, there are two different processes of rank x and y. Suppose there is a communicator, produced by MPI_Comm_split applied to MPI_COMM_WORLD, with the name NEWCOMM. Is there a possibility that each process, x and y, broadcasts a value to NEWCOMM (for example process x broadcasts integer value 10 and process y broadcasts float value 3.14) but none of these two processes receive each other's value? If yes, explain how. If no, explain why. Assume no deadlock happens. The original number of processes in MPI_COMM_WORLD is not relevant to the solution of this question.

[3] Yes, this can happen.
[3] When processes x and y call MPI_Comm_split, they used different *colors* and hence, NEWCOMM for process x is a different communicator than NEWCOMM for process y.

(b) [6 points] Suppose process x sends two messages to process y. That is, process x executes two consecutive MPI_Send; and process y executes two consecutive MPI_recv. We know that MPI_Recv is blocking. Also, there may be a probability that the two messages sent by process x arrive to process y out of order (i.e. the second message arrives first).

- Why the messages may arrive out of order?
- Does process y face a problem of being blocked forever if messages arrive out of order? Justify your answer.

[2] [If you mention a different reason that makes sense, you get the credit for that part]. One reason messages can arrive out of order is that each message took a different route in the interconnection network and one of the routes has more contention.

[2] Process y does not face any problems.
[2] MPI run time buffers the messages. When the message belonging to the first MPI_Recv arrives, it will be delivered to process y. If the second message arrives first, it will be buffered till the first message arrives and is delivered. Also, another way of dealing with that is for process y to use the wildcard MPI_ANY_TAG to receive whichever message arrives first. [If you mention one reason that makes sense you get the credit for that part].

(c) [6 points] If each process in a communicator call MPI_Reduce twice. That is, two reduction operations one after the other. Can some of the processes finish the first reduction operation and move to the next one before the other processes finish theirs? Justify. This question is not related to the questions (a) and (b) above.

[2] This cannot happen.
[4] Because reduction operation is a collective call and is blocking. So the process will be blocked until all others processes are done with the first reduction before they all move to the next one.

(d) [4 points] Why do MPI processes not suffer from cache coherence overhead?

Because they do not share any address space. Hence the virtual addresses generated by a process will never produce physical addresses similar to other processes. For example, virtual address 0x10000 from process 0 will be translated to a different physical address than the one produced from the same virtual address 0x10000 from process 1.

(e) [8 points] If we have four processes in MPI generated with mpiexec -n 4. Each one has a fraction F of its code that is inherently sequential and cannot be parallelized. The sequential code in each process is the same in all the four processes. Using Amdahl's law, what is the theoretical speedup we get from using four processes over one process, assuming F = 20%? Show all your steps to get full credit.

Since they have been generated from the same process, and F is the same part(s) of the code in all of them, then it is a traditional Amdahl's law problem.
Speedup(4) = 1 / (0.2+(0.8/4)) = 1/0.4 = 2.5
[Partial credits:
Find that it is just using Amdahl's law: 2 points
Citing the law: 2 points
Filling correct numbers: 2 points
Final correct answer: 2points
However, if the final answer is correct and the steps make sense, even if missing something from the above, the student gets full credit.]