COMS4040A and COMS7045A Class Test

April 10, 2018

Student Number:

Instructions

1. This is a closed book class test. Some designated reference materials are provided and can be used.
2. The test includes four parts. Hons students are requested to complete Parts I, II, and III. MSc students are requested to complete all four parts.
3. For the questions require written answers, write them in the spaces provided in the test script.
4. Put all your solution programs in a compressed file named by your student number, then submit it on sakai. In the case the sakai submission fails, email it to **hairongwng@gmail.com**.
5. All the written answers and codes must be done for clarity and readability. When it comes to programming, proper comments should be added where necessary.
6. Duration: 3 hours.

Total Mark: [ ]/50

Part I Questions [8 marks]

1. (a) Use task/channel model to design a parallel algorithm for simulating 2D heat conduction problem. A possible finite difference approximation is as follows.

$$\frac{\Delta T}{\Delta t} = \alpha(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}),$$

where ΔT is the temperature change over time Δt, T is temperature, α is a constant, and i, j are the indices in the grid. In the beginning, the boundary points of the grid are assigned with initial temperature and the inner points update their temperature iteratively. Each inner point updates its temperature by using the previous temperature of its neighbouring points and itself. Draw task/channel diagrams to illustrate your design. [6]

(b) Give a complexity analysis of your parallel algorithm in Item 1a. [2]

Part II OpenMP Programming [21 marks]

1. Consider the loop

```
a[0]=0;
for(i=1; i<n; i++)
  a[i]=a[i-1] + i;
```

There is clearly a loop-carried dependence, as the value of `a[i]` can't be computed without the value of `a[i-1]`. Can you see a way to eliminate this dependence so that the loop can be parallelized? [3]

2. If $A = (a_{i,j})$ is an $m \times n$ matrix and x is a vector with n components, then $y = Ax$ is a vector with m components. We can find the ith component of y by forming the dot product of the ith row of A with x:

$$y_i = a_{i,0} \times x_0 + a_{i,1} \times x_1 + a_{i,2} \times x_2 + \cdots + a_{i,n-1} \times x_{n-1}.$$

(a) Based on the serial implementation of matrix vector multiplication given in `mat_vect.c`, implement a parallel solution using OpenMP. You only need to modify `mat_vect` function in `mat_vect.c`. Type `make` to compile your program. Run your program with different sets of data and differing number of threads indicated in Table 1 (or simply type `./run.sh`, the provided script to run), and record your run time in Table 1. [6]

	$m \times n$		
	8000000×8	8000×8000	8×8000000
Threads	Time (s)	Time (s)	Time (s)
1			
4			
8			

Table 1: Matrix-vector multiplication input specifications

(b) After completing the problem above, the program was ran on a laptop with 2 physical cores (each core has its own cache). The results are shown in Table 2. The metric 'E' in Table 2 represents the

	$m \times n$					
	8000000×8		8000×8000		8×8000000	
Threads	Time	E	Time	E	Time	E
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.3000	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Table 2: Matrix-vector multiplication run-time and efficiencies for different input data sets

efficiency of a parallel program. It shows the utilization of threads in an OpenMP program, and computed as $E = S/t$, where S is the speedup and t is the number of threads. Based on the results in Table 2, answer the following questions. Note that the total number of floating point additions and multiplications in each case is 64,000,000.

i. What may cause the different running times for the three input data sets while we are using the same number of threads? Elaborate your answer. [3]

ii. What may cause the poor efficiency for the input 8×8000000 when more than 1 thread is used? Elaborate your answer. [3]

3. The program `linked.c` traverses a linked list and computes a sequence (starting from 38) of Fibonacci numbers at each node. Parallelize this program using OpenMP. Type `make` to compile and type `./run.sh` to run your program. [6]

Part III CUDA C/C++ Programming [23 marks]

1. Complete the program `gray2bw.cu` by implementing a kernel function, which performs the following image transform:

$$g(x,y) = \begin{cases} 1 & \text{if } f(x,y) > 0.5 \\ 0 & \text{if } f(x,y) \leq 0.5. \end{cases}$$

   You are requested to complete the kernel function `transformKernel()` only. Once you complete the program, type `make` to compile and type `./run.sh` to run. [6]

2. Listings 1 and 2 present two slightly different algorithms for *sum* reduction using shared memory. (Note that the algorithm is incomplete.)

   a) What is the difference between these two algorithms? [2 marks]

   b) Is there a performance difference between these two kernels? Why? [3 marks]

c) Implement both kernels. A base program `cuda_reduction.cu` is provided. For simplicity, we consider the case of an array of only 1024 elements in this program. Thus, you should launch your kernel with one block of 1024 threads only. These are declared as macros in the base program

```
#define N 1024

#define NUM_THREADS_PER_BLOCK 1024.
```

The base program given is entirely a serial code. To complete the program, besides implementing the two kernels, you also need to add into the base program the declarations of device variables, their memory allocations, data transfers between host and device etc.. Once you complete the program, type `make` to compile and type `./run.sh` to run. [12 marks]

```
__shared__ float partialSum[SIZE];
partialSum[threadIdx.x] = X[threadIdx.x];

unsigned int t = threadIdx.x;
for(int stride = 1; stride < blockDim.x; stride *= 2)
{
  __syncthreads();
  if(t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
```

Listing 1: Kernel A

```
__shared__ float partialSum[SIZE];
partialSum[threadIdx.x] = X[threadIdx.x];

unsigned int t = threadIdx.x;
for(int stride = blockDim.x/2; stride >= 1; stride = stride >> 1)
{
  __syncthreads();
  if(t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

Listing 2: Kernel B

Part IV [10 marks]

1. The Sieve of Eratosthens can be used to find all the prime numbers in a list of natural numbers. The serial algorithm is given in Algorithm 1. For example, in order to find the primes up to 100, we mark all the multiples of 2 first. We then go on to mark all the multiples of 3, 5 and 7. When we come to mark the multiples of 11, we stop since $11^2 > 100$. All the unmarked numbers represent the primes in the list.

   How would you parallelize this algorithm for a shared memory programming model such as OpenMP? Discuss your approach with regard to data decomposition and data communication among the threads. [10]

Algorithm 1: The Sieve of Eratosthens to find primes between 2 and n.

```
Input: A list of natural numbers
Output: The unmarked list of primes
1 begin
2     Set $k$ to 2, the first unmarked number in the list
3     Repeat
        a) Mark all multiples of $k$ between $k^2$ and $n$
        b) Find the smallest number greater than $k$ that is unmarked.
        c) Set $k$ to this new value
      Until $k^2 > n$
      The unmarked numbers are prime
```