**CS 133 Parallel and Distributed Computing**
**Winter 2014**
**Jason Cong**

# Homework #2
# Due Jan. 23, 2014

**Reading assignment:**
Lecture notes 3 and 4.

**Homework problems:**

1. For the prefix sum problem discussed in lecture 3, please come up with a non-recursive parallel algorithm to compute all prefix sums. Please analyze the complexity of your algorithm in terms of the total work and running time.

2. Please do the following assignment using OpenMP with either C or C++. The assignment has three parts.

**Part 1:**

The following function can be used to determine if a given integer is prime or not. If the integer is prime, the function returns 1, else a 0.

```
int is_prime (int p) {
   int i;

   if (p < 2) return 0;
   i = 2;
   while (i*i <= p) {
      if (p % i == 0) return 0;
      i++;
   }
   return 1;
}
```

Using the code of this function, write an OpenMP program that finds all the prime numbers from 2 to given N. The numbers should be evenly divided among threads, which test the primality of each number. For instance, the loop could be broken down such that thread 0 is currently testing the primality of 2, thread 1 is testing the primality of 3, thread 2 is testing the primality of 4, and so on.

The threads will be performing some redundant work, but this should be ok. Also, there will be a workload imbalance, which is addressed in Section 2.

**Input:** N, T

N is the maximum number, and T is the number of threads.

The inputs should be passed in command line. For example, if your program is compiled to an executable named hw2a, the following command line will be used:

$ ./hw2a N T

Use OpenMP's `omp_set_num_threads(…)` function to learn how many threads are specified at command line.

**Output:**

Your program should output all the prime numbers between 2 and N separated by commas, and finishes by a newline.

Sample output (N=10):

```
2, 3, 5, 7
```

Your output numbers must be in ascending order, as the sample output. The format must also follow the exact format, because the results will be evaluated by automatic scripts.

**Part 2:**

Adapt the code from section 1 to use the `schedule` clause of OpenMP. This should address workload imbalance by allowing new tasks to be assigned to threads as they become idle. The syntax is `schedule (dynamic [, chunk-size])`

If not specified, the chunk-size is 1. Begin with this.

**Part 3:**

You are going to write a Monte-Carlo approximation of PI, using the producer-consumer framework with OpenMP. The algorithm works by generating a set of random points inside a square. Then, the number of points that are also inside the square's inscribed circle is counted. Based on Monte-Carlo method, the ratio between the number of points inside the circle and the number of total points approximates the ratio between the area of the circle and the square. So the value of PI can be approximated this way. The method illustrated by the following pseudo code:

```
int num_in = 0;
for (int i=0; i<N; i++) {
    double x = rand()/RAND_MAX;          // Producer, 0<=x<=1
    double y = rand()/RAND_MAX;          // Producer, 0<=y<=1
    double r = sqrt((x-0.5)*(x-0.5) + (y-0.5)*(y-0.5))   // Consumer, distance to (0.5, 0.5)
    if (r<=0.5) num_in++;                // Consumer, if r<0.5 the point is inside the circle
}
PI = 4*num_in/N;
```

You need to implement a program with `sections/section` to create a single producer task and a single consumer task: (also marked in the pseudo code above)

Producer: generate random points inside the square from (0, 0) to (1, 1).

Consumer: counts the points inside the square's inscribed circle.

The producer and consumer communicate through a shared a FIFO, which is implemented as a circular buffer using an array of size 32. The FIFO holds the coordinates of the points (you can both create a struct with two floating-point numbers or use two separate arrays). The producer thread adds a new point at the tail of the array and he consumer thread reads a point from the head of the buffer. If the array is empty, the consumer waits for the producer, which can be implemented as a while loop. Similarly, when the buffer is full the producer needs to wait for the consumer. The total number of points generated by producer is given as command line input, and the consumer should consume the same amount of points.

You must use task parallel (`sections`) for this problem.

**Input**: N

N is the number of points used in the approximation. Please keep in mind this N could be larger than the shared buffer size. The input should be passed in command line. For example:

$ ./hw2c N

**Output**:

Your program should output a single floating-point number of the approximate PI produced by your program, followed by a newline. You must use AT LEAST seven digits after the decimal point.

Sample output:

3.152356

**Hints**:

You should use the proper locks when using shared objects.

At the end of your program, make sure the circular buffer is empty.

Run your program multiple times, as the behavior of such a parallel task is dynamic at runtime.

**Questions related to Problem 2:**

1) Run the code from Part 1 on 1 thread, 2 threads, 4 threads, 8 threads, and 16 threads. Use N = 20,000,000. Record the total execution time of the program using the UNIX `time` utility.

```
$ time program-name
```

2) Run the code from Part 2 on 1, 2, 4, 8, and 16 threads using N = 20,000,000. How much better does dynamic scheduling work here? Could you explain the reason for the results?

Then, adjust the chunk-size of the dynamic and test the code again on 16 threads. Are you able to make a significant difference with chunk-sizes larger than one? Record the trend in execution time as the chunk-size is set to 2, 4, 8, 16, 32. Could you explain the reason for the results?

3) For the problem of section 3, please explain the potential hazards when two threads are trying to access the shared resource. If you encounter any problem during development, please give an example and explain how you resolved it.

4) (Bonus Question) The method used in Part 1 to calculate prime number is very inefficient. Can you improve the algorithm using the sift method? Parallelize your method using OpenMP, and try your best to optimize the performance. How does it compare with the original algorithm?

## What to turn in

Submit to Courseweb a single tarball (with extension .tar.gz or .tgz) containing a directory named

Inside this directory should reside the following files:

- Source code for all completed portions…
  - For section 1, a single file named hw2a.c (or hw2a.cpp if using C++)
  - For section 2, a single file named hw2b.c (or hw2b.cpp if using C++)
  - For section 3, a single file named hw2c.c (or hw2c.cpp if using C++)
- An ASCII file named hw2answers.txt containing the answers to all questions in section 3
  - NOTE: do **not** submit any other format, such as Word .doc/.docx, OpenOffice, PDF etc

## Grade criteria

- Submission instructions were followed to the letter: 10%
  - (Yes…you get 10% just for submitting a tarball with the files/structure asked for, regardless of the file contents.)
- Source code compiles without error on cs133.seas.ucla.edu: 30%
  - 10% per section
- Compiled code executes and provides correct output: 30%
  - 10% per section
- Questions answered: 30%
  - 10% per question

## Late penalties

Late submissions will be penalized by 10% per day, and accepted up to 1 week after the due date.

## Lab notes

For this programming homework, please use cs133.seas.ucla.edu for debugging and profiling. This machine has been reserved for our class, and can accommodate 16 simultaneous threads. Please start your project as early as possible to avoid overload on the machine. The gcc and g++ compilers on the machine can compile OpenMP code with the –fopenmp command-line switch.

You may develop your code elsewhere, but you **must** ensure that it works on cs133.seas.ucla.edu, which is where it will be graded. Please answer all questions using executions on cs133.seas.ucla.edu.