CSC 410/510 Programming Assignment #1 – Due 18 September

1. The sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to any given limit.

   To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

   - Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., *n*).
   - Initially, let *p* equal 2, the first prime number.
   - Starting from *p*, enumerate its multiples by counting to *n* in increments of *p*, and mark them in the list (these will be 2*p*, 3*p*, 4*p*, etc.; the p itself should not be marked).
   - Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.

   When the algorithm terminates, all the numbers in the list that are not marked are prime.

   Create a parallel algorithm that finds all primes less than *n* (where *n* is provided by the user and is fairly large, about 1 to 10 million or so, but feel free to use smaller *n* to make sure your algorithm works correctly) on *p=8* processes with shared memory and prints a list of them to the screen. Time your code using both `static` and `dynamic` schedule to determine if there is any difference in performance and if your algorithm is deterministic. The output should provide a list of all primes less than *n* together with the runtime.

   Example of output:

```
./prime 1000000

    0: 2 3 5 7 11 13 17 19 23 29
   10: 31 37 41 43 47 53 59 61 67 71
...
...
78480: 999721 999727 999749 999763 999769 999773 999809 999853 999863 999883
78490: 999907 999917 999931 999953 999959 999961 999979 999983
Elapsed time = 8.042866e+00 ms
```

   Yes I know, I made 2 the 0th prime, but I am a computer scientist. There are several web-pages out there that provide lists of primes. Either as all the primes below a given number, or 1000, 10000, 100000 etc. first primes. So you should have no problem finding a way to check your list.

2. Suppose we toss darts randomly at a square dartboard whose sides are 1 ft in length. Suppose also that there is a quarter circle segment with radius 1 ft inscribed in the square dartboard, from the top left corner, to the lower right corner. If the points that are hit by the darts are uniformly distributed (and always landing in the square), then the number of darts that hit inside the circle segment versus the total number of darts thrown should approximately satisfy a quarter of π:

$$\frac{\pi}{4} = \frac{darts\ inside\ cicle\ segment}{total\ number\ of\ darts\ tossed}$$

   This is a so called Monte Carlo process, as it uses randomness to solve a problem.

Write an *OpenMP* program that uses the Monte Carlo method to estimate π. Read in the total number of tosses before forking any threads. Display the results after joining all threads. Use `long long int`for the number of tosses and hits in the circle, get a reasonable estimate (that is correct to 5 to 6 decimal places) of π. Also notice that `rand` is not particular random, but `random` has its own problem of not being thread-safe. This problem can be overcome, and I am sure you can figure out how.

Things you should be able to answer:

- How many darts are needed to reach a reasonable estimate?
- Is there any measurable difference between using a critical statement and using a reduction clause?
- Time your code using both `static` and `dynamic` schedule to determine if there is any difference in performance?

Assignment Submission:

Your homework is due at the beginning of class. Tar or zip all documentation and source files together. The header of each source file should contain all the normal information (name, class assignment etc.)

- Documentation - all of the following should be in: `prog1.pdf`
  - Description of the programs.
  - Description of the algorithms and libraries used.
  - Description of functions and program structure.
  - How to compile and use the program.
  - Description of the testing and verification process.
  - Description of what you have submitted: `Makefile`, external functions, main, etc.
  - Format: PDF (write in any word processor and export as PDF if the option is available, or convert to PDF)
- Main programs
- Any required external functions *.c.
- Any include files you use (other than the standard ones).
- The `Makefile` to build the programs
- Tar the directory and then gzip using the correct filename.

[And I do know that there are better compression routines than gzip.]