Programming assignment #2: CSE 100 Spring 2013

Your assignment is to implement a multithreaded program that tests for primality. The assignment will provide experience in programming parallel multicore processors, using the thread facility of C++. This assignment is a starting point for exploring multithreading, and doesn't cover more advanced topics such as synchronization. Your code will serve as a departure point for writing your own applications to use the extra core(s) on your own hardware.

In addition to learning about threads, this assignment will also familiarize you with command line argument processing, and the `vector` container template.

>>> Due Fri Apr 26 at 8:00 PM.

>>> Required solution files: Primes.cpp, getInput.cpp, prime_thrFn.cpp, Report.cpp

The files that contain your solution to the assignment must be turned using the bundleP2 script by the due deadline. Please read BundleP2's instruction. If you have trouble downloading the two files, you can download them here: SIGNATURE and DECLARATION

If you are working in a team of 2 students, turn in your assignment from *one* of your two accounts only. In your source code files, clearly document the names and login names of the authors. You may switch partners or work in the same team as the previous assignment.

In doing this assignment, it's your responsibility to understand the course rules for integrity of scholarship.

Getting Started

Read the assigned materials from the Williams text, which are provided to you on the course website. Make sure you understand the basics of the thread and vector interfaces.

Create a directory named P2 under the home directory for your class account on ieng6. Your files should be bundled from your `P2` directory using the `bundleP2` script before the due time for the assignment.

The code in `~/../public/P2/` is a starter code for the primality test program discussed in class. This code will not run correctly until you've added certain code described below and it doesn't support multithreading. Copy over the provided files into your newly created P2 directory.

Assignment

As discussed in class, your code will read in a list of numbers from the command line, and test whether or not the numbers are prime, in parallel on a multicore processor. (Our hardware testbeds have 4 cores.) The provided code contains some of the components needed to construct your code but will not run correctly as is and it doesn't spawn threads.

Your coding should be done in two steps. The first step is to produce a working serial (single core) program in preparation for parallelizing it in step 2.

Step 1: The serial code

First, modify `getInput()` to read the candidate primes from the command line into a C++ `vector`. Since

vectors resize themselves automatically, you won't need to allocate any storage. The prime number input vector is declared for you as a global variable (i.e., outside the scope of any function) in preparation for parallelizing the code in step 2. You'll note that the thread function declares this vector as a global via the `extern` keyword.

Next, modify the provided thread function to access the candidate primes from the prime number vector instead of the command line. The thread function will set the `PrimeFlags` vector to `true` for each number that passes the primality test: there is one flag array per candidate prime.

Lastly, write a tester to print out the prime numbers, and note how many are prime. This tester should be called `ReportPrimes()`, and it must use the signature provided in `Primes.cpp`. We will use our own tester to check your program, so do not modify the signature of `ReportPrimes()`.

In the course of making the above changes, you'll also need to make changes to the main driver routine, Primes.cpp, consistent with the above modifications.

Once you have restructured the code, test it with the provided input files, primes_8.sh and primes_64.sh. All of the numbers specified in these shell scripts are prime. Don't attempt to parallelize the code until you have a correctly running serial program.

Step 2: The parallel code

Once your serial program is correct, parallelize it to run on multiple cores with C++ threads. A template for parallel programs is provided in `~/../public/Threads`, that shows you how to create and join threads. Make sure you test that your program will work on a single thread once you set up the fork/join code.

Next, modify `getInput()` to read the number of threads from the command line (it should come before the list of candidate primes). Then, modify the thread function so that it assigns each thread a contiguous section of the `numbers` vector. Your code should correctly handle the case when the number of threads (*NT*) doesn't divide into the amount of numbers to be tested (*N*) evenly, but don't attempt this option until you can correctly handle the simple case where they do.

To run your program, you'll use the following synopsis, which will test 4 candidate primes numbers on 2 cores (all but 42 are prime)

```
./primes 2 42 34155013283297 34155013283401 34155013283437
```

The corresponding output appears in the file `ExampleOutput.txt` Note the final line of output:

```
#> 4 3 2 0.286516
```

This line summarizes all the performance information of the run and is a handy way to extract the output from many runs. This particular string identifies a run that tested 4 prime numbers, found 3 to be prime and ran on 2 cores in 0.286516 seconds. The leading string "#>" can be used as a search string, for example, using he `grep` search command.

Since you will be using multiple cores in this assignment, you may experience stronger contention (interference) from other users, as you all compete for the available cores. To reduce this contention we will use additional resources made available by ACMS: `ieng6-250` through `ieng6-254.` To minimize the chance of unevenly machine load, we will divide up the class according to the first letters of your last name. Details will be announced on Piazza.

Despite the measures we'll use to reduce contention, we cannot avoid it entirely. Thus, you should run your program a minimum of 10 times, and use the *minimum* timing. Note that, if you only observed the minimum one time out of 10, run for 5 more iterations until you observe that minimum at least twice (to within 5% to 10%).

Documentation

Document your code. For this assignment, you will not turn in a separate user's manual; comments within the source code files themselves will suffice, if they are clear, informative, complete, and appropriately written.

Describe how to use your program, what its purpose is, and what its limitations are. Be sure to indicate who wrote the code; both students' names and login names must appear prominently in all source code files, if the assignment was done as a team.

Though you aren't required to turnin a speedup report, we advise that you create a file showing speedups as a function of the number of threads, so you can check that your code is scaling properly.

Notes and Hints

You should test your program as thoroughly as you can (we will certainly test it as thoroughly as we can when grading it). To check that your code is correct, use a web site to test the prime numbers, e.g. `http://www.math.com/students/calculators/source/prime-number.htm`

If you plan to use the gdb, make sure you use the following command: make gdb=1. Be sure to do a `make clean` before using the `gdb=1` option, or if you want to stop compiling with debugging output turned on.

Note that the code times the entire computation from the time the threads are spawned to the time they are joined. Don't time any things else, or you may get misleading timings. In particular, be sure to remove or comment out any I/O in areas of the code that are bracketed by the calls to the timer. You should observe excellent speedups (nearly perfect, to within 5% to 10%) provided you give each core a few seconds worth of work; this will effectively render the cost of creating and joining threads to be a relatively small fraction of the total running time.

The code uses 64 bit integer test numbers. This permits good running time when you do your data collection.

Grading

There are 25 possible points on the assignment. If your solution files do not compile and link error-free, you will receive 0 points for this assignment. We will compile the files you turn in with the command `make`.

A good submission should compile, and it should obtain perfect, or near perfect speedups up to 4 cores (within 10% is OK) when presented with a list of large candidate primes as in the provided file `run64.sh`. To get full credit, style and comments count.