Prof. Jingke Li (lij@pdx.edu); Classes/Labs: TR 1400-1550 @FAB 150, 88-10; Office Hours: TR 1300-1400 @FAB 120-06

# Assignment 2: More Shared-Memory Programming

# (Due Thursday, 5/5/22)

This assignment continues to practice shared-memory programming. This time, you will write several versions of a prime-finding program, and collect timing data to analyze their performance. All programs will use C++. One of the programs is optional for CS415 students (see details below).

## 1. Base Version (`prime.cpp`)

The file `prime.cpp` contains a sequential implementation of the sieve of Eratosthenes prime-finding algorithm. The program reads in a command-line argument, `N`, and finds all prime numbers up to the limit `N`. It starts with the first prime, 2, marking all its multiples up to `N` as composites. It then moves to the next prime, 3, which is the first unmarked number after 2. This process continues until all primes within the *sieve range* $[2..\sqrt{N}]$ are found, and their multiples marked. All remaining unmarked numbers in `[2..N]` are now primes.

**Your Task** Insert timing routines into the program to measure its elapsed time. Include everything in the timing brackets, except for the command-line input processing, and the print statements (which appear at both ends of the program). Report timing in units of millisecond. Here is a sample output:

```
linux> ./prime 1000
prime (seq) over [2..1000] starting ...
prime (seq) found 168 primes in 0.0239 ms
```

## 2. OpenMP Version (`prime-omp.cpp`)

Copy the timing-augmented program `prime.cpp` to `prime-omp.cpp`. There are three loops (one of them is a loop-nest) in the program. All are parallelizable. However, for the loop nest, you need to decide with loop of the nest is parallelizable.

**Your Tasks**

1. Insert directives to parallelize all parallelizable loops. Add necessary clauses to the directives. (*Hint:* How to handle the updates to the variable `totalPrimes`?)
2. Modify the program to accept an optional second commend-line argument, `P`. Use `P` to set the number of threads to use in all parallel regions. If a user does not provide a value for `P`, use a default value of 1.
3. Modify the program's print statements, so that the messages will match the program, e.g.

```
linux> ./prime-omp
Usage: ./prime-omp N [P]
linux> ./prime-omp 1000 0
P must be greater than 0
linux> ./prime-omp 1000
prime-omp (1 threads) over [2..1000] starting ...
prime-omp (1 threads) found 168 primes in 0.3343 ms
```

## 3. Naive Multi-Thread Version (`prime-par1.cpp`)

Implement a naive multi-thread version of prime-finding algorithm in `prime-par1.cpp`. (You may copy `prime.cpp` over to use as a starting point.) This naive version works quite similarly to the OpenMP program. It runs the master first to find all sieve primes (in the range `[2..`$\sqrt{N}$`]`); it then runs the workers concurrently, each for a section of the range `(`$\sqrt{N}$`..N]`. This program does not need synchronization, except for updating the global count `totalPrimes`. Here is a more detailed description of the algorithm:

**Algorithm for `prime-par1.cpp`:**

1. [*Params*] Master receives values of `N` (and possibly `P`) from command-line arguments.
2. [*Init*] Master allocates arrays `candidate[N]` and `sieve[`$\sqrt{N}$`]`.
3. [*Sieve*] Master (a) initializes (only) the sieve section `[2..`$\sqrt{N}$`]` of `candidate[]`; (b) it finds all sieve primes in the sieve section and save them in `sieve[]`; and (c) it adds the total number of sieves to the global variable `totalPrimes`.
4. [*Start-Threads*] Master creates `P` threads, `worker[0]..worker[P-1]`.
5. [*Worker*] Each worker (a) figures out its corresponding section in the range `(`$\sqrt{N}$`..N]`, and initializes that section of `candidate[]`; (b) it walks through the sieve primes in `sieve[]`, and marks off their multiples in its section of `candidate[]`; (c) it tallies the primes found in its section, and adds the number to `totalPrimes`.
6. [*End-Threads*] Master waits for workers to join back, and prints out the final `totalPrimes` result.

**Requirements**

- Your program should strictly follow the given algorithm.
- Use the same user interface as the OpenMP version, i.e. one required parameter (`N`) and one optional parameter (`P`, with default value of 1).
- Implement the program using C++'s thread support (i.e. no Pthread routines).
- The range `(`$\sqrt{N}$`..N]` is evenly partitioned into `P` sections. (*Hint:* You should insert debug print statements to verify the partition.)
- Each worker is responsible for all work related to its section, including the initialization of the section in the `candidate[]` array.

## 4. Proper Multi-Thread Version (`prime-par2.cpp`)

Develop a second multi-thread version of the prime-finding program, and save it in `prime-par2.cpp`. From algorithm point of view, this new version is almost the same as the previous version – the only difference is that Steps 3 and 4 of the algorithm are switched; as a result, the master's sieve-finding action runs concurrently with the worker threads. (*Hint:* You should complete `prime-par1.cpp` first, before starting on this version.)

Here are two new issues to handle:

- [**Synchronization**] Due to the concurrent execution of the master thread and the worker threads, there is a need for synchronization. The master needs to notify workers after each new sieve is found, and correspondingly, the workers need to wait for new sieves to appear, if they had run ahead of the master.
- [**Termination**] Since the master does not produce all sieves before the workers start, there is a need for the workers to dynamically check for a termination condition. One idea is to have the master set a global variable `totalSieves` after it has completed the sieve-finding action, and have the workers check its sieve index against that variable periodically.

**Requirement** You should use condition variables for synchronization and termination checking. No busy-waiting (i.e. repeated checking with a loop) is allowed.

## 5. A Third Multi-Thread Version (`prime-par3.cpp`)

In the previous two versions, the program's workload (represented by the range `(`$\sqrt{\texttt{N}}$`..N]`) is static partitioned into `P` parts. An alternative approach is to use dynamic partitioning, i.e. let the workers compete for new tasks. Here is the new algorithm.

**Algorithm for `prime-par3.cpp`:**

1. [*Params*] Master receives values of `N` (and possibly `P`) from command-line arguments.
2. [*Init*] Master allocates arrays `candidate[N]` and `sieve[`$\sqrt{\texttt{N}}$`]`; it also initializes the whole `candidate[]` array.
3. [*Start-Threads*] Master creates `P` threads, `worker[0]..worker[P-1]`.
4. [*Sieve*] Master works over the range `[2..`$\sqrt{\texttt{N}}$`]` and finds all sieves, and save them in the `sieve[]` array; after each sieve is found, it notifies the workers.
5. [*Worker*] All workers compete to get the next sieve from `sieve[]`, with a shared global index; the winning worker updates the index and goes to mark off the sieve prime's multiples in the whole range `(`$\sqrt{\texttt{N}}$`..N]` of the `candidate[]` array; the workers terminate when there is no more sieves to work on.
6. [*End-Threads*] Master waits for workers to join back, and prints out the final `totalPrimes` result.

**Note:** Steps 4 and 5 are concurrent.

**Requirements**

- Implement this algorithm in `prime-par3.cpp`.
- Timing measurements are still required for this program.
- Each worker also keeps track of the number of sieve primes it worked on; save the info in a `stats[P]` array. (Note that since each thread is accessing only its own cell, there is no need to guard the updates to this array.)

## 6. A Second OpenMP Program* (`prime-omp2.cpp`)

[* For CS415 students, this part is optional; if you correctly implement it, you will earn an extra 15% of points.]

In this version, we would like to emulate the algorithm used in `prime-par3.cpp`, i.e. assign threads to work on individual sieve primes, rather than on sections of the range `(`$\sqrt{\texttt{N}}$`..N]`.

**Your Tasks**

1. Rewrite the sequential program `prime.cpp` to make it work for this algorithm. You are allowed to split loops and to introduce new variables. But your modification should not substantially change the program's total workload, e.g. there should be no introduction of redundant computation. (*Hint:* Think of how to separate the sieve-finding action (to be performed by the master) from the action of using a sieve to process the numbers in the range of `(`$\sqrt{\texttt{N}}$`..N]` (to be performed by workers).)
2. Insert directives so that threads will work on individual sieve primes, as specified by the algorithm.

## 7. Timing Study

Collect timing results from all programs. Varying `N`'s value from 1000 to as high as the system allows, with an increment of `10x`, and varying `P`'s value from 1 to 128, with an increment of `2x`. To get the best results, you should collect timing on the 45-CPU server machine `babbage.cs.pdx.edu`. Tabulate and analyze the results like you did in Lab 4. Discussing observations and drawing conclusions as you see fit. In particular, document and discuss any unexpected performance behaviors. (*Hint:* You may want to write a shell script to help automating the running of these programs. Sample scripts are available in the Assignment 1 package.)

## Summary and Submission

Write a one- or two-page summary covering your experience with this assignment, including your timing study in it. If you have noticed any program hanging or other abnormal behaviors (e.g. uneven workload distribution), document them in your summary. Make a zip file containing all your programs and your write-up, and submit it through the “Assignment 2” folder on Canvas.

**Grading Metrics** Correctness and conformance to requirements are the main metrics for grading. Points are roughly evenly distributed over all units of this assignment, with Sections 1 and 2 counts as one unit.