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, 5/9/24)

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

This assignment carries a total of 20 points, evenly divided over the required programs (two for CS415 students, three for CS515 students), with 2 points reserved for the report.

Prime-Finding Algorithm

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 first unmarked number after the current prime, which will be the next prime (in this case 3). This process continues until all primes within the *sieve range* `[2..`$\sqrt{\texttt{N}}$`]` are found, and their multiples marked. All remaining unmarked numbers in `[2..N]` are now primes.

You are to implement three multi-thread versions of this algorithm in C++, `prime-par[123].cpp`. These programs share the same user interface,

```
linux> prime-parX N [P]   // P = #threads (excluding master)
```

i.e. they take in two command-line arguments, one required and one optional: `N` for the search limit, and `P` for the number of worker threads. If the second argument is missing, `P` takes the default value 1.

1. First Multi-Thread Version (`prime-par1.cpp`)

Implement a multi-thread version of prime-finding algorithm with C++'s thread library. Call the new program `prime-par1.cpp`. (You may copy `prime.cpp` over as a starting version.) This version runs the master first to find all sieve primes (in the range `[2..`$\sqrt{\texttt{N}}$`]`); it then runs the workers concurrently, each for a section of the range `(`$\sqrt{\texttt{N}}$`..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 two arrays, `bool candidate[N]` and `int sieve[`$\sqrt{\texttt{N}}$`]`. The latter is for holding sieve primes: `sieve[0]=2`, `sieve[1]=3`, etc.
3. [*Sieve Primes*] Master finds the sieve primes:
   (a) it initializes (only) the sieve section `[2..`$\sqrt{\texttt{N}}$`]` 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. [*Worker Threads*] Master creates `P` threads, `worker[0]..worker[P-1]`.
5. [*Worker's Task*] Each worker performs the following work:
   (a) it figures out its unique section in the range `(`$\sqrt{\texttt{N}}$`..N]`, and initializes that section of `candidate[]`;
   (b) for each prime in `sieve[]`, it marks off its multiples in its section of `candidate[]`; and
   (c) it tallies the primes found in its section, and adds the number to `totalPrimes`.
6. [*Ending*] Master waits for workers to join back, and prints out the final `totalPrimes` result.

Requirements

- Your program should strictly follow the given algorithm.
- Implement the program using C++'s thread support (i.e. no Pthread routines).
- The range `(`$\sqrt{\texttt{N}}$`..N]` is evenly partitioned into `P` sections. (If `P` does not evenly divide the range's size, assigning the extra items to the last worker is fine.)
- Each worker is responsible for all work related to its section, including the initialization of the section in the `candidate[]` array. And it should only work within its section.
- Master and workers should print out the `candidate` array section they worked on, and the number of primes they found. The appendix section at the end shows a sample output, which you can use as a reference.

2. Second 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 (Step 3) runs concurrently with the worker threads (Step 5). You should complete `prime-par1.cpp` 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 prime is found, and correspondingly, the workers need to wait for new sieve primes to appear, if they had run ahead of the master.
- [**Termination**] Since the master does not produce all sieve primes 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.

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

3. Third Multi-Thread Version (`prime-par3.cpp`)

[* For CS415 students, this part is optional; if you correctly implement it, you will earn an extra 20% of points.]

In the previous two versions, the program's workload (represented by the range `(`$\sqrt{\texttt{N}}$`..N]`) is static partitioned into `P` sections. An alternative approach is to use dynamic partitioning, i.e. let the workers compete for new tasks. With this approach, the candidate range `(`$\sqrt{\texttt{N}}$`..N]`) is no longer partitioned. Instead, each worker is responsible for marking of multiples of a selected sieve prime in the whole range.

In addition to synchronization and termination, there is one new issue. For eacher worker, once a sieve prime is selected, it needs to mark off its multiples in the range `(`$\sqrt{\texttt{N}}$`..N]`). For that, it needs to figure out the `first` such multiple in the range, so that it can execute the following loop:

```
for (int i = first; i <= N; i += p)
  candidate[i] = false;
```

(*Hint:* Try to find a mathematical formula for calculating `first`.)

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. (The last action is different from the previous algorithm.)
3. [*Worker Threads*] Master creates `P` threads, `worker[0]..worker[P-1]`.

4a. [*Sieve Primes*] Master finds all sieve primes in the sieve section `[2..`$\sqrt{\texttt{N}}$`]` and save them in `sieve[]`; after each sieve prime is found, it notifies the workers.

4b. [*Worker's Task*] All workers compete to get the next sieve from `sieve[]`, which is pointed to by 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 sieve primes to work on.

5. [*Ending*] Master waits for workers to join back, and prints out the final `totalPrimes` result.

Note: Steps 4a and 4b are concurrent.

Requirements

- Your program should strictly follow the given algorithm. In particular, the loop for marking off multiples in Step `4b` should start with a index value in the range `(`$\sqrt{\texttt{N}}$`..N]`. In other words, not to duplicate master's work in the sieve range.
- 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.)
- Master should print out the number of sieve primes it found. The workers should print out the number of primes they worked on. The appendix section at the end shows a sample output, which you can use as a reference.

4. 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.

Appendix. Sample Outputs

```
linux> ./prime-par1 10000 4
prime-par1 (4 threads) over [2..10000] ...
Master found 25 primes in [2..100]
```

```
Worker[0] found 350 primes in [101..2575]
Worker[1] found 300 primes in [2576..5050]
Worker[2] found 278 primes in [5051..7525]
Worker[3] found 276 primes in [7526..10000]
prime-par1 (N=10000,P=4) found 1229 primes in 0.693439 ms

linux> ./prime-par3 10000 4
prime-par3 (4 threads) over [2..10000] ...
Master found 25 primes in [2..100]
Worker[3] worked on 6 primes
Worker[0] worked on 6 primes
Worker[2] worked on 8 primes
Worker[1] worked on 5 primes
prime-par3 (N=10000,P=4) found 1229 primes in 1.30157 ms
```