Prof. Jingke Li (FAB 120-06, lij@pdx.edu); Class: MW 14:00-15:15 @ FAB 47; Lab: F 10:00-11:30 @ FAB 88-10.

Assignment 2: More Shared-Memory Programming

(Due Wednesday, 2/13/19)

This assignment continues to practice shared-memory programming. This time, you will use both OpenMP and Pthreads to implement multiple versions of a prime-finding algorithm. This assignment carries a total of 20 points.

415 vs 515: CS515 students are required to complete all five programs. CS415 students are required to complete four programs, the first three plus one of the last two of their choice.

1. Parallelizing a Simple Prime-Finding Algorithm

The file `prime.c` contains a sequential implementation of the sieve of Eratosthenes prime-finding algorithm. The program reads in a command-line argument, `N`, and finds all primes up to `N`. It starts with the first prime number, 2, marking all its multiples up to `N` as composites. Then it moves to the next prime, 3, which is the first unmarked integer 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.

Now think about how to parallelize this algorithm. Looking inside the program, `prime.c`, we can see that the main loop nest:

```
for (int i=2; i<=limit; i++)
  if (array[i])
    for (int j=i+i; j<=N; j+=i)
      array[j] = 0;
```

contributes the most to the program's execution time. The outer loop iterates through the sieve range, and the inner loop is for marking the multiples of a given sieve prime.

We therefore can focus on just parallelizing this loop nest, and ignore the rest of the program. There are two general parallelization strategies.

1. **Parallelizing the outer loop.** Ideally, we want to assign one thread for each sieve prime, rather than one for each number in the sieve range, and have it be responsible for marking all of the multiples of that prime. For example, if we have four threads available, we can assign them to the first four primes, 2, 3, 5, and 7, and have them marking the multiples of their prime. Then they can work on the next group of primes, and so on. One challenge with this strategy is that not all sieve primes are available at the start of the program; in fact, only one is known. At any time, the next prime can only be found after the marking of some multiples of the previous primes. In other words, there are ordering constraints between the earlier threads and the new thread for the next prime. This means there has to be some synchronization among the threads.
2. **Parallelizing the inner loop.** With this approach, the data range `[1..N]` is evenly partitioned into `P` sections, and each section is assigned to a separate thread. Each thread is responsible for marking multiples of all sieve primes within its section. For instance, if $\texttt{P}_x$ is assigned the section `[101..200]`, then it will be responsible for marking all the composites in that section.

Your Task is write one OpenMP and four Pthread parallel prime-finding programs, based on these two strategies. All five programs share the same user interface:

```
linux> ./prime-xxx N [P]
```

where `N` is the prime-searching bound, and `P` represents the number of threads. (`P` is optional; if not provided, it assumes the default value of 1.)

2. An OpenMP Version

Copy `prime.c` to a new file `prime-omp.c`. Your task is to implement Strategy 2, *i.e.* parallelizing the inner loop, with OpenMP directives. Here are the specific requirements:

- Set `P` as the number of threads to use for the parallel region.
- Use the routine `omp_get_thread_num()` to get the thread ID within the parallel region.
- For each thread, keep track of how many composites it has marked.
- Report the results, in a similar format as the following:

```
linux> ./prime-omp 100 4
Seaching primes in [1..100] with 4 threads
-- W[0] working on prime 2 (1st composite:4)
-- W[3] working on prime 2 (1st composite:78)
-- W[1] working on prime 2 (1st composite:30)
-- W[2] working on prime 2 (1st composite:54)
-- W[2] working on prime 3 (1st composite:54)
-- W[1] working on prime 3 (1st composite:30)
...
Found 25 primes in [1..100]: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97, ...
Thread[0]:30
Thread[1]:28
Thread[2]:28
Thread[3]:27
Total: 113
```

Note: The above printouts show evidence that the four threads are working on four different sections, since their first composites are 4, 30, 54, and 78, respectively. You need to insert a `printf` statement into the parallel region to generate the expected output. (It can be a little tricky to set control for not printing too many messages, *i.e.* printing just the first composite for each prime. If you can't do it, it's fine.)

3. Four Pthread Versions

For this second part, you are going to write four Pthread programs: two for implementing Strategy 1 and two for implementing Strategy 2. In all programs, the master thread is responsible for finding all the sieve primes, and the worker threads are for marking composites. Each worker thread should track the number of composites it has marked, and the master thread should report the statistics at the end, just like in the OpenMP case.

The file `prime-pthd0.c` contains a partial implementation, and can be used as a starting template for all programs. You should keep the program structure and the print statements. Additional print statements may be needed for some programs. The programs' output should look similar to the samples, which are included at the end of this handout.

- The first program, **prime-pthd1.c**, implements Strategy 1. It works as follows:
  1. Master initializes a candidate array, `array[N]`.
  2. Master finds all sieve primes, and save them in an array, `sieve[]`. (*Note:* For this step, all markings should be confined within the sieve range `[2..`$\sqrt{\texttt{N}}$`]`.)
  3. Master creates `P-1` threads, `worker[1], ..., worker[P-1]`; master itself becomes `worker[0]`.

4. Each worker competes to get the next sieve from `sieve[]`, and marks its multiples in `array[]`.
5. Each worker terminates itself when there is no more sieve to work on.
6. Master waits for other workers to finish, and prints out the results.

- The second program, **prime-pthd2.c**, implements Strategy 2. It requires that `P` evenly divides `N`. The program works as follows:

  0. Master checks that `P` evenly divides `N`; if not, it rejects the parameters and exits.
  1.-3. (Same as `prime-pthd1.c`)
  4. Divide the data range `[1..N]` into `P` even sections, one for each worker. Each worker iterates through all the primes in `sieve[]`, and marks their multiples within its data section.
  5. Master waits for other workers to finish, and prints out the results.

- The third program, **prime-pthd3.c**, is an improved version of `prime-pthd1.c`. In this version, instead of having all workers waiting for the master to find all the sieves first, the master runs the `find_sieves()` routine *concurrently* with the worker threads. It works as follows:

  1. Master initializes a candidate array, `array[N]`.
  2. Master creates `P-1` threads, `worker[1], ..., worker[P-1]`.
  3. Master finds all sieve primes, and save them in an array, `sieve[]`. After that, master itself becomes `worker[0]`.
  4. Each worker competes to get the next sieve from `sieve[]`, and marks its multiples in `array[]`.
  5.-6. (Same as `prime-pthd1.c`)

  The program structures of `prime-pthd1.c`) and `prime-pthd3.c` are almost identical, except that Steps 2 and 3 are switched. Here, the worker threads are created first.

  Note: Steps 3 and 4 *must* run concurrently. To ensure that, you may consider adding a line,

  ```
  usleep(10000); // sleep for 10ms to make sure workers are ready
  ```

  between Steps 2 and 3. (Need to include the header `unistd.h` for this to work.)

  This program is more challenging. With the concurrency, you need to find a way for the master to signal the workers every time a new sieve is found. You also need to find a way to for the workers to know when to terminate.

- The forth program, **prime-pthd4.c**, is an improved version of `prime-pthd2.c`, with the same concurrency setup between master's `find_sieves()` and the worker threads, as in `prime-pthd3.c`. This program still requires that `P` evenly divides `N`. It works as follows:

  0. Master checks that `P` evenly divides `N`; if not, it rejects the parameters and exits.
  1. Master initializes a candidate array, `array[N]`.
  2. Master creates `P-1` threads, `worker[1], ..., worker[P-1]`.
  3. Master finds all sieve primes, and save them in an array, `sieve[]`. After that, master itself becomes `worker[0]`.
  4. Divide the data range `[1..N]` into `P` even sections, one for each worker. Each worker iterates through all the primes in `sieve[]`, and marks their multiples within its data section.
  5. Master waits for other workers to finish, and prints out the results.

  Note: Steps 3 and 4 *must* run concurrently. Again, you need to think about how to synchronize the threads.

Summary and Submission

Save an execution script for each of your programs. Write a short (one-page) summary (in plain text or pdf) covering your experience with this assignment. Make a zip file containing all your programs, the execution scripts, and your write-up. Submit it through the "Assignment 2" dropbox on D2L.

Sample Output

```
linux> ./prime-pthd1 100 4
Seaching primes in [1..100] with 4 threads
Master found 4 sieves
Worker[1] starts ...
Worker[2] starts ...
Worker[3] starts ...
-- W[1] working on prime 2
-- W[1] working on prime 7
-- W[2] working on prime 3
Worker[2] done
Worker[0] starts ...
Worker[0] done
Worker[1] done
-- W[3] working on prime 5
Worker[3] done
Found 25 primes in [1..100]: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97, ...
Thread[0]:0
Thread[1]:62
Thread[2]:32
Thread[3]:19
Total: 113

linux> ./prime-pthd2 100 4
Seaching primes in [1..100] with 4 threads
Master found 4 sieves
Worker[1] starts on range [26..50] ...
Worker[0] starts on range [1..25] ...
-- W[0] working on prime 2 (1st composite:4)
-- W[0] working on prime 3 (1st composite:6)
-- W[0] working on prime 5 (1st composite:10)
-- W[0] working on prime 7 (1st composite:14)
Worker[0] done
Worker[3] starts on range [76..100] ...
-- W[3] working on prime 2 (1st composite:76)
-- W[3] working on prime 3 (1st composite:78)
-- W[3] working on prime 5 (1st composite:80)
-- W[3] working on prime 7 (1st composite:77)
Worker[3] done
-- W[1] working on prime 2 (1st composite:26)
-- W[1] working on prime 3 (1st composite:27)
-- W[1] working on prime 5 (1st composite:30)
-- W[1] working on prime 7 (1st composite:28)
Worker[1] done
Worker[2] starts on range [51..75] ...
-- W[2] working on prime 2 (1st composite:52)
-- W[2] working on prime 3 (1st composite:51)
-- W[2] working on prime 5 (1st composite:55)
-- W[2] working on prime 7 (1st composite:56)
Worker[2] done
Found 25 primes in [1..100]: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97, ...
Thread[0]:24
Thread[1]:30
Thread[2]:29
Thread[3]:30
Total: 113
```