CPSC 457 Winter 2023 - Assignment 3

Due date is posted on D2L.
Individual assignment. Group work is NOT allowed.
Weight: 21% of the final grade.

For both programming questions you need to write multithreaded code. You may use pthreads, or C++ threads.

Q1. Programming question – calculating π [10 marks]

Improve the performance of an existing single-threaded `calcpi` program by converting it to a multi-threaded implementation.

Download the starter code, compile it, and run it:

```
$ git clone https://gitlab.com/cpsc457w23/pi-calc.git
$ cd pi-calc
$ make
$ ./calcpi
Usage: ./calcpi radius n_threads
   where 0 <= radius <= 100000
     and 1 <= n_threads <= 256
```

The `calcpi` program estimates the value of π using an algorithm described in:

https://en.wikipedia.org/wiki/Approximations_of_%CF%80#Summing_a_circle's_area.

Most of the algorithm is implemented inside the function `count_pixels()` in file `calcpi.cpp`. The included driver (`main.cpp`) parses the command line arguments, calls `count_pixels()` and prints the results. The driver takes 2 command line arguments: an integer radius and number of threads. For example, to estimate the value of π using radius of 10 and 2 threads, you would run it like this:

```
$ ./calcpi 10 2
Calling count_pixels(r=10, n_threads=2)...
Result = 317 pixels (estimated PI=3.17)
```

The function `uint64_t count_pixels(int r, int N)` takes two parameters – the radius and number of threads. It then returns the number of pixels inside the circle with radius r and centered at (0,0), by checking every pixel (x, y) in squre $-r \leq x, y \leq r$. The current implementation is single threaded, so it ignores the 2nd argument `N`. Your job is to re-implement the function so that it uses `N` threads to speed up its execution, such that it runs `N` times faster with `N` threads on hardware where `N` threads can run concurrently. Please note that your code will be marked both for correctness and for the speedup it achieves. In order for your code to be considered correct, your multi-threaded implementation needs to return the **same** number of pixels as the single-threaded implementation.

For this question, you are **only allowed to create and join threads**. You need to find a way to parallelize `count_pixels()` without using any synchronization mechanisms, such as mutexes, semaphores, atomic types, etc.

Write all code into `calcpi.cpp` and submit this file for grading. Make sure your `calcpi.cpp` works with the included driver program. We may use a different driver program during marking, so it is important that your code follows the correct API. Make sure your program runs on `linuxlab.cpsc.ucalgary.ca`.

Assume $0 \leq r \leq 100{,}000$ and $1 \leq nthreads \leq 256$.

Timing your code on linuxlab machines

Please note that not all `linuxlab` machines are the same. Some have 6 physical cores with hyper-threading enabled, while others have 8 physical cores, but no hyperthreading. When you are running your timings, please make sure you do all of them on the same machine. Otherwise, you will get inconsistent results. You can check which CPU you have by running `lscpu` command.

My basic multi-threaded implementation achieves the following timings using `r=100000`. I expect your solutions to achieve similar results.

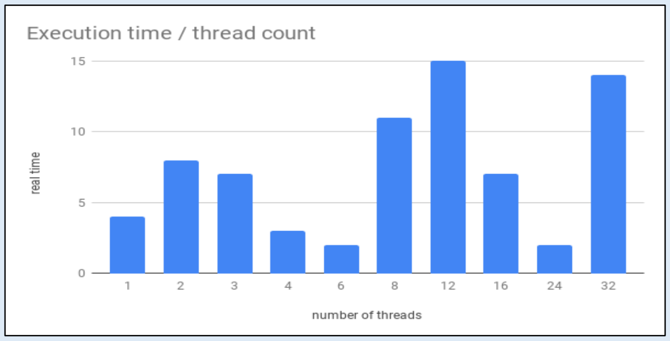
CPU	1 thread	2 threads	4 threads	8 threads	16 threads
i7-8700	10.67	5.56	2.84	1.61	1.21
i7-9700	10.62	5.5	2.82	1.36	1.36

Q2. Written answer [3 marks]

Time your multi-threaded solution from Q1 with $r = 50000$ using the `time` command on `linuxlab.cpsc.ucalgary.ca`. Record the real-time for 1, 2, 3, 4, 6, 8, 12, 16, 24 and 32 threads. Also record the timings of the original single-threaded program.

A. Record your timings in a table and create a corresponding bar graph. Format the table and the graph like the ones below (the numbers below are random and your timings should look different).

Threads	Timing (s)
1 (original)	5
1	4
2	8
3	7
4	3
6	2
8	11
12	15
16	7
24	2
32	14



B. When you run your implementation with N threads, you should see N-times speed up compared to the original single threaded program. Do you observe this in your timings for all values of N?

C. Why do you stop seeing the speed up after some value of N?

Q3. Programming question – detecting primes [30 marks]

For this part of the assignment, you must convert a single-threaded program `detectPrimes` to a multi-threaded implementation.

Start by downloading the single-threaded code, then compile it and run it:

```
$ git clone https://gitlab.com/cpsc457w23/detect-primes.git
$ cd detect-primes
$ make
$ cat example.txt
      0   3   19 25 3
4012009 165 1033

$ ./detectPrimes 5 < example.txt
Using 5 threads.
Identified 4 primes:
  3 19 3 1033
Finished in 0.0000s

$ seq 100000000000000000 100000000000000300 | ./detectPrimes 2
Using 2 threads.
Identified 9 primes:
  100000000000000003 100000000000000013 100000000000000019 100000000000000021
  100000000000000049 100000000000000081 100000000000000099 100000000000000141
  100000000000000181
Finished in 5.6863s
```

The `detectPrimes` program reads integers in range $[2, 2^{63} - 2]$ from standard input, and then outputs the ones that are prime numbers. The first invocation example above detects prime numbers 3, 19, 3 and 1033 in a file `example.txt` (notice that number 3 is repeated both in the input and output). The second invocation uses the program to find all primes in the range $[10^{17}, 10^{17} + 300]$. If duplicate primes appear in the input, they will be duplicated in the output.

`detectPrimes` accepts a single command line argument – a number of threads. This parameter is not used in the current implementation, as the starter code is single threaded. Your job is to improve the execution time of `detectPrimes` by making it multi-threaded, and your implementation should use the number of threads given on the command line. To do this, you will need to re-implement the code in `detectPrimes.cpp`, namely the function:

```
std::vector<int64_t>
detect_primes(const std::vector<int64_t> & nums, int n_threads);
```

The function takes two parameters: the list of numbers to test, and the number of threads to use. It returns the prime numbers found, in arbitrary order. The function is called by the driver (`main.cpp`) after parsing the standard input and command line. Your implementation should use `n_threads` number of threads. Ideally, if the original single-threaded program takes time T to complete a test, then your multi-threaded implementation should finish that same test in T/N time when using N threads. For example, if it takes 10s to complete a test for the original single-threaded program, then it should take your multi-threaded program only 2.5s to complete that same test with 4 threads. To achieve this goal, you will need to design your program so that:

- You give each thread the same amount of work;
- your multi-threaded implementation does the same amount of work as the single-threaded version; and
- the synchronization mechanisms you utilize are efficient.

We will mark your assignment by running the code against multiple different inputs and using different numbers of threads. To get full marks for this assignment, your program needs to output correct results but also achieve near optimal speedup for the given number of threads and available cores. If your code does not achieve optimal speedup on all inputs, you will lose some marks for those tests.

You may assume that there will be no more than $100{,}000$ numbers in the input, and that all numbers will be in the range $[2, 2^{63} - 2]$. Some inputs will include many numbers, some inputs will include just few numbers, some numbers will be large, some small, some will be prime numbers, others will be large composite numbers, etc… For some numbers it will take long time to compute their smallest factor, for others it will take very little time. You need to take all these possibilities into consideration. Design your own test inputs and test your code thoroughly.

Write all code into `detectPrimes.cpp` and submit it for grading. Make sure your `detectPrimes.cpp` works with the included `main.cpp` driver. We may use a different driver during marking, so it is important that your code follows the correct API. Make sure your program runs on `linuxlab.cpsc.ucalgary.ca`.

You may use any of the synchronization mechanisms we covered in lectures, such as semaphores, mutexes, condition variables, spinlocks, atomic variables, and barriers. Make sure your code compiles and runs on `linuxlab.cpsc.ucalgary.ca`.

Please note that the purpose of this question is NOT to find a more efficient factorization algorithm. You **must** implement the **exact same** factorization algorithm as given in the skeleton code, except you need to make it multi-threaded.

Q4. Written question (5 marks)

Time the original single-threaded `detectPrimes.cpp` as well as your multi-threaded version on three files: `medium.txt`, `hard.txt` and `hard2.txt`. For each of these files, you will run your solution 6 times, using 1, 2, 3, 4, 8 and 16 threads. You will record your results in 3 tables, one for each file, formatted like this:

`medium.txt`			
# threads	**Observed timing**	**Observed speedup compared to original**	**Expected speedup**
original program		1.0	1.0
1			1.0
2			2.0
3			3.0
4			4.0
8			8.0
16			16.0

The 'Observed timing' column will contain the raw timing results of your runs. The 'Observed speedup' column will be calculated as a ratio of your raw timing with respect to the timing of the original single-threaded program. Once you have created the tables, explain the results you obtained. Are the timings what you expected them to be? If not, explain why they differ.

Submission

Submit the following files to D2L for this assignment:

`calcpi.cpp`	solution to Q1
`detectPrimes.cpp`	solution to Q3
`report.pdf`	answers to all written questions

Please note – you need to **submit all files every time** you make a submission, as the previous submission will be overwritten. **Do not submit a ZIP file. Submit individual files.**

General information about all assignments:

- All assignments are due on the date listed on D2L. Late submissions will not be marked.
- Extensions may be granted only by the course instructor.
- After you submit your work to D2L, verify your submission by re-downloading it.
- You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
- Assignments will likely be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
- All programs you submit must run on `linuxlab.cpsc.ucalgary.ca`. If your TA is unable to run your code on the Linux machines, you will receive 0 marks for the relevant question.
- **Assignments must reflect individual work**. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
- Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions with anyone else; you are not allowed to sell or purchase a solution. This list is not exclusive.
- We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.

Appendix – Hints for Q1

I suggest you parallelize the outer loop. Give each thread roughly equal number of columns in which to count the pixels. Then sum up the counts from each thread. Your overall algorithm could look like this:

- Create separate memory area for each thread (for input and output), e.g.
  ```
  struct Task { int start_x, end_x, partial_count; ...};
  Task tasks[256];
  ```
- Divide the work evenly between threads, e.g.
  ```
  for(int i = 0 ; i < n_threads ; i ++) {
    tasks[i].start_x = … ;
    tasks[i].end_x = … ;
  }
  ```
- Create threads and run each thread on the work assigned to it. Each thread counts pixels for the x-range assigned to it and updates its `partial_count`.
- Join the threads.
- Combine the results of each thread into final result – i.e. return the sum of all `partial_counts`.

Appendix – Hints for Q3

Hint 1 – bad solution (do not implement this)

A bad solution would be to parallelize the outer loop of the algorithm and assign a fixed portion of the numbers to each thread to check. This is a terrible solution because it would not achieve speedups on many inputs, for example where all hard numbers are at the beginning, and all the easy ones at the end. Your program would then likely give all hard numbers to one thread and would end up running just as slowly as a single-threaded version.

Hint 2 – simple solution (start with this)

A much better, yet still simple solution, would be to parallelize the outer loop, but instead of giving each thread a fixed portion of the input to test, each thread would dynamically determine how many numbers it would process. For example, each thread could be setup to process the next number in the list, and if it is a prime, it would add it to the result vector. This would repeat until all numbers have been tested. All you need to implement this solution is a single mutex to guard access to the input vector, and to the result vector. I strongly suggest you start by implementing this simple solution first, and only attempt the more difficult approaches after your simple solution works.

Note that this solution would achieve optimal speedup for many inputs, but not for all. For example, on input with a single large prime number, it will not achieve any speedup at all. Consequently, if you choose this approach, you will not be able to receive full marks for some tests.

Hint 3 – good solution

Even more efficient approach is to parallelize the inner loop (the loop inside the `is_prime` function). In this approach, all threads would work on testing the same number for primality. If you choose this approach, you need to give each thread a different portion of divisors to check. This will allow you to handle more input cases than the simple solution mentioned earlier. For extra efficiency, and better marks, you should consider implementing thread re-use, e.g., by using barriers. Here is a possible rough outline of an algorithm that you could implement:

```
detectPrimes():
  prepare memory for each thread
  initialize empty array result[] - this could be a global variable
  set global_finished = false - make it atomic to be safe
  start N threads, each runs thread_function() on its own memory
  join N threads
  return results

thread_function():
  repeat forever:
    serial task - pick one thread using barrier
      get the next number from nums[]
      if no more numbers left:
        set global_finished=true to indicate to all threads to quit
      otherwise:
        divide work for each thread
    parallel task - executed by all threads, via barrier
      if global_finished flag is set, exit thread
      otherwise do the work assigned above and record per-thread result
    serial task - pick one thread using barrier
      combine the per-thread results and update the result[] array if necessary
```

The only synchronization primitive you should need to implement this hint is a barrier. You should not need to use any other synchronization primitives (remember that any code inside your serial code does not require to be protected by mutexes…). You may use my C++ barrier implementation from lecture notes if you wish.

Hint 4 – best solution

This builds on top of the hint 3 above, by adding to it thread cancellation. You need cancellation for cases where one of the threads discovers the number being tested is not a prime, so that it can cancel the work of the other threads. I suggest using a single atomic boolean variable to implement the cancellation flag. Please note that while thread cancellation is very simple to implement, it does take non-trivial effort to get it working correctly.

Appendix – Approximate grading scheme for Q3

The test cases that we will use for marking will be designed so that you will get full marks only if you implement the most optimal solution. However, you will receive partial marks even if you implement one of the less optimal solutions. Here is the rough breakdown of what you can expect depending on which solution you implement:

- Parallelization of outer loop with fixed amount of work (hint 1) will yield ~9/28 marks

- Parallelization of outer loop with dynamic work (hint 2) will yield ~15/28 marks
- Parallelization of inner loop (hint 3) without work cancellation and without thread reuse will yield ~19/28 marks
- Parallelization of inner loop (hint 3) with thread reuse but without cancellation will yield ~24/28 marks.
- Parallelization of inner loop (hint 4) with thread reuse and with cancellation will give 28/28 marks.

Please note that any tests on which your program produces wrong results will receive 0 marks.