CPSC 457 - Assignment 3

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

Q1. Programming question - multithreaded π calculation [10 marks]

For this question you will improve the performance of an existing program by converting it from single-threaded implementation to a multi-threaded implementation. Download starter code, compile it and run it:

```
$ git clone https://gitlab.com/cpsc457/public/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 here:

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

The algorithm is implemented inside function `count_pi()` in file `calcpi.cpp`. The included driver program `main.cpp` parses the command line arguments, calls `count_pi()` and prints the results. The command line arguments are radius `r` and number of threads `n_threads`. For example, to estimate the value of π using radius of 10 and a 2 threads, you would invoke it like this:

```
$ ./calcpi 10 1
Calculating PI with r=10 and n_threads=1
count: 317
PI:    3.17
```

The function

```
uint64_t count_pi(int r, int n_threads)
```

takes two parameters – the radius and number of threads, and returns the number of pixels inside the circle or radius r centered at $(0,0)$ for every pixel (x, y) in squre $-r \leq x, y \leq r$. The current implementation is single-threaded. Your job is to re-implement the function so that it uses `n_threads` threads to speed up its execution. If you implement it correctly, the `calcpi` program should run N times faster with N threads, providing the hardware can run N threads concurrently. Your TAs will mark your code both on correctness and the speedup you achieve.

You can use the pthread library or `std::thread` for this question. You are **not allowed** to use any synchronization mechanisms for this question – such as mutex, semaphores, atomic types, etc. I suggest you to take the following approach:

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

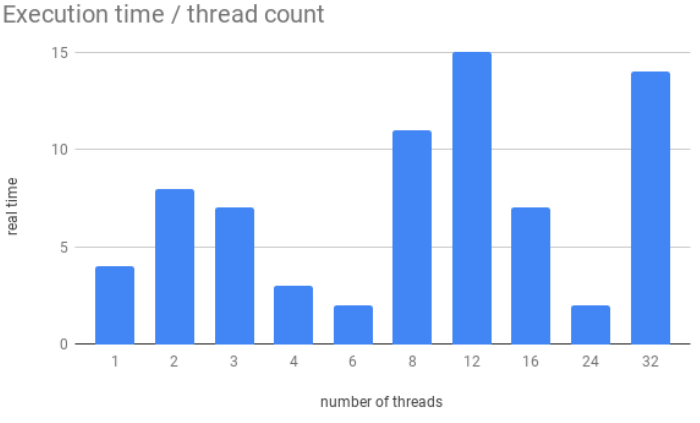
Write all code into `calcpi.cpp` and submit this file for grading. Make sure your `calcpi.cpp` works with the included driver program. Your TAs may use a different driver program during marking, so it is important that your code follows the correct API. Make sure your program runs on `linuxlab.cpsc.ucalgary.ca`.

You can assume that `r` will be in range [0..100,000] and `n_threads` will be in range [1..256].

Q2 – Written answer [3 marks]

Time your multi-threaded solution from Q1 on 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 for the original single-threaded program. In your report, include a table of these timings, and a bar graph, both formatted similar to these:

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



Please note that the numbers in the above table and graph are random; your timings should look diffferent.

Answer the following questions:

a) 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 N?
b) Why do you stop seeing the speed up after some value of N?

Q3. Programming question - multithreaded factor sum [30 marks]

This question is similar to Q1 – you will convert a single-threaded program `sumFactors` to a multi-threaded implementation, in order to improve its performance. The `sumFactors` program reads integers in range [0..2^63-2] from standard input. For each number it reads, it ignores any numbers that are smaller than 2 or prime numbers. For the remaining composite numbers, it calculates their smallest non-trivial factors (≥ 2). The program sums up all these factors and prints the final sum to standard output.

Download the single-threaded code here, compile it and run it on `example.txt` file with 1 thread:

```
$ git clone https://gitlab.com/cpsc457/public/factor-sum
$ cd factor-sum
$ make
$ cat example.txt
      0   3   19 25
4012009 165 1033
$ ./sumFactors 1 < example.txt
Using 1 thread.
Sum of divisors = 2011
```

Number 0 is ignored becase it is smaller than 2. Numbers 3, 19 and 1033 are ignored because they are prime numbers. Smallest non-trivial factors of 25, 4012009 and 165 are 5, 2003 and 3, respectively, and their sum is $5 + 2003 + 3 = 2011$.

The program accepts a single command line argument – a number of threads. This parameter is currently not used in the implementation. Your job is to improve the execution time of `sumFactors` by making it multi-threaded, by using the requested number of threads. To do this, you will need to re-implement the function:

```
int64_t sum_factors(int n_threads);
```

which is defined in `sumFactors.cpp`. The function takes a single parameter `n_threads` – the number of threads to use. 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 10 seconds to run a test for the original single-threaded program, then it should take your multi-threaded program only 2.5 seconds to run it with 4 threads. In order to achieve this goal, you will need to design your program so that:

- each thread is doing roughly equal amount of work for all inputs; and
- the synchronization mechanisms are efficient.

Your TAs will mark your assignment by running the code against multiple different inputs and using different number of threads. To get full marks for this assignment, your program needs to:

- output correct results; and
- achieve the optimal speedup for the given number of threads and available cores.

If your code does not achieve optimal speedup on all inputs, you will not receive full marks. 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 of these possibilities into consideration.

A bad solution would be to read in all numbers first, and then give a portion of the numbers to each thread to check. The reason this is a bad solution is that I could then carefully craft an input that has all the hard numbers to check at the beginning, and all the easy ones at the end. Your program would then likely give all of the hard numbers to one thread, and your program would run just as slowly as a single threaded version.

A slightly better solution would be to parallelize your code so that each thread processes the next number of the input, records the found factor, and then moves onto the next number. This solution would work for many cases, but not for all cases. For example, if the input only contains a single number, your parallel solution will not speed up at all. If you choose this approach, you will not be able to receive full marks for some of the tests.

A more difficult approach involves parallelizing the code so that all threads process the same number, i.e. you would parallelize the `get_smallest_divisor()` function. If you choose this approach, you need to give each thread a different portion of factors to check. This will allow you to handle more cases than the simple solution mentioned earlier. However, you need to be careful to make sure you don't introduce race conditions. Also, you will need to think about how to handle thread cancellation in case one of the threads disovers a small factor. I strongly suggest that you start by implementing the simple solution first, and only attempt the more difficult solution when your simple solution already works.

Write all code into `sumFactors.cpp` and submit this file for grading. Make sure your `sumFactors.cpp` works with the included driver program. Your TAs 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`.

You may assume that there will be no more than 10,000 numbers in the input, and that all numbers will be in the range [0 .. 2^63-2].

Please note that the purpose of this question is NOT to find a more efficient factorization algorithm. The purpose of this question is to parallelize the existing solution, using the exact same factorization algorithm given in `sumFactors.cpp`.

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

Q4 – Written question (5 marks)

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

Test file: `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 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
`sumFactors.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.

General information about all assignments:

1. All assignments are due on the date listed on D2L. Late submissions will be not be marked.
2. Extensions may be granted only by the course instructor.
3. After you submit your work to D2L, verify your submission by re-downloading it.
4. 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.
5. Assignments will 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.
6. 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.
7. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on linuxlab.cpsc.ucalgary.ca, when compiled with `-O2` optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
8. **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.
9. 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.
10. 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 1 – grading scheme for Q3

- Parallelization of outer loop with fixed amount of work will yield ~9/28 marks
- Parallelization of outer loop with dynamic work will yield ~15/28 marks
- Parallelization of inner loop without work cancellation and without thread reuse will yield ~19/28 marks
- Parallelization of inner loop with thread reuse (eg. barriers) but without cancellation will yield ~24/28 marks.
- Parallelization of inner loop with thread reuse and with cancellation will give 28/28 marks.