CPSC 457 Winter 2020 - Assignment 3

Due date is posted on D2L.
Individual assignment. Group work is not allowed!
Weight: 5% of the final grade.

Q1 – Written question (5 marks)

Suppose a program spends 2/3 of time doing I/O. Calculate the CPU utilization when running 5 copies of such a program at the same time. Assume one single-core CPU, and all I/O operations are executed in parallel.

Q2 - Programming question – multithreaded (30 marks)

In Appendix 1 you will find a C program called `countPrimes.c`, which reads numbers from the standard input, and counts how many of them are prime. The appendix includes instructions on how to compile and run the program.

You are free to use any parts of the `countPrimes.c` in your program. However, you are **not allowed** to use any other code in your solution that you did not write yourself (unless it was explicitly provided for you by the intstructor or your TA).

Your job is to improve the execution time of `countPrimes.c` by making it multi-threaded, using pthreads. Your multithreaded program should take a single command line argument 'N', which will determine how many threads your program will be allowed to use. Ideally, if the original program takes time T to complete a test, then your program 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 program, then it should take your 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; 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… You need to take all of these possibilities into consideration. Your TAs will provide some hints for this question in the tutorials.

The output of your program must match the output of the original program exactly. 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 prime test algorithm. The purpose of this question is to parallelize the existing solution, using the exact same prime testing algorithm.

Q3 – Written question (5 marks)

Time the original program as well as your solution 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 looking 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, eplain the results you obtained. Are the timings what you expected them to be? If not, explain why they differ.

Submission

You should submit 2 files to D2L for this assignment:

`report.[pdf|txt]`	answers to all written questions
`count.[c|cpp]`	solution to Q2 in C or C++

General information about all assignments:

1. All assignments must be submitted before the due date listed on the assignment. Late assignments or components of assignments will not be accepted for marking without approval for an extension beforehand. What you have submitted in D2L as of the due date is what will be marked.
2. Extensions may be granted for reasonable cases, but only by the course instructor, and only with the receipt of the appropriate documentation (e.g. a doctor's note). Typical examples of reasonable cases for an extension include: illness or a death in the family. Cases where extensions will not be granted include situations that are typical of student

life, such as having multiple due dates, work commitments, etc. Forgetting to hand in your assignment on time is not a valid reason for getting an extension.

3. After you submit your work to D2L, make sure that you check the content of your submission. It is your responsibility to do this, so make sure that you submit your assignment with enough time before it is due so that you can double-check your upload, and possibly re-upload the assignment.
4. All assignments should include contact information, including full name, student ID and tutorial section, at the very top of each file submitted.
5. Assignments must reflect individual work. Group work is not allowed in this class nor can you copy the work of others. 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.
6. You can and should submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It's better to submit incomplete work for a chance of getting partial marks, than not to submit anything.
7. 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.
8. All programs you submit must run on the CPSC Linux machines. If your TA is unable to run your code on the Linux machines, you will receive 0 marks for the relevant question.
9. TAs will expect good code design. You should deal with bad inputs, give useful messages if program/scripts are mis-used, use exit codes, and have well commented code.

The conversion between a percentage grade and letter grade is as follows.

	A+	A	A-	B+	B	B-	C+	C	C-	D+	D
Min % Req	96%	92%	88%	84%	80%	76%	72%	68%	64%	60%	55%

Appendix 1 – countPrimes.c

```
/// counts number of primes from standard input
/// compile with:
///   $ gcc countPrimes.c –O2 -o count –lm
/// by Pavol Federl, for CPSC457 Spring 2017, University of Calgary
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>

/// primality test, if n is prime, return 1, else return 0
int isPrime(int64_t n)
{
     if( n <= 1) return 0; // small numbers are not primes
     if( n <= 3) return 1; // 2 and 3 are prime
     if( n % 2 == 0 || n % 3 == 0) return 0; // multiples of 2 and
     int64_t i = 5;
     int64_t max = sqrt(n);
     while( i <= max) {
         if (n % i == 0 || n % (i+2) == 0) return 0;
         i += 6;
     }
     return 1;
}

int main( int argc, char ** argv)
{
    /// parse command line arguments
    int nThreads = 1;
    if( argc != 1 && argc != 2) {
        printf("Uasge: countPrimes [nThreads]\n");
        exit(-1);
    }
    if( argc == 2) nThreads = atoi( argv[1]);
    /// handle invalid arguments
    if( nThreads < 1 || nThreads > 256) {
        printf("Bad arguments. 1 <= nThreads <= 256!\n");
    }
    if( nThreads != 1) {
        printf("I am not multithreaded yet :-(\n");
        exit(-1);
    }
    /// count the primes
    printf("Counting primes using %d thread%s.\n",
           nThreads, nThreads == 1 ? "s" : "");
    int64_t count = 0;
    while( 1) {
        int64_t num;
        if( 1 != scanf("%ld", & num)) break;
        if( isPrime(num)) count ++;
    }
    /// report results
    printf("Found %ld primes.\n", count);
    return 0;
}
```

To compile this program, you can use a C or C++ compiler:

```
 $ gcc countPrimes.c -O2 -o count -lm
 $ g++ countPrimes.c -O2 -o count -lm
```

When you run the program from the terminal, it will wait for you to type in the numbers on standard input. To indicate EOF you press <ctrl-d>:

```
$ ./count
Counting primes using 1 threads.
1 3 <enter>
5 <enter>
<ctrl-d>
Found 2 primes.
```

You can make testing easier by preparing different input files and then use standard input redirect ‘<’ to feed a file to the program. For example, you can store 4 numbers in a file `test.txt`:

```
123 321 13
11
```

To run the code on the file, instead of having to type in the numbers yourself, you can:

```
$ ./count < test.txt
Counting primes using 1 threads.
Found 2 primes.
```

In the above example the program found 2 primes (13 and 11). You can find additional test files in Appendix 2.

You are free to use any parts of the `countPrimes.c` above in your program. However, you are **not allowed** to use any other code in your solution that you did not write yourself (unless it was explicitly provided for you by the intstructor or your TA).

Appendix 2 – test files for countPrimes.c

easy.txt

```
1 2 3 4 5 6 7 8 9 10
100 101 102 103 104 105
106 107
```

medium.txt

```
9007199254740997 8796451843471 2 900719925474105073 90071996392800899
4 90071992547410493 90071992547410613 900719925474105161 900719925474104987
1 90072054854210657 900719925474104927 9007199254741049 900720046819564759
90071992547410513 9007207298190743 9007206203142577 5 3 900719925474105013
9007199254740881 8796254359103 90071992547410663 9007199254741033
900719925474104977 90071992547410609 900720004735038979 8796093022151
```

hard.txt

```
1 2 3 4
9223372036854775783
5 6 7 8 9
```

hard2.txt

```
9223371873002223329 1 2 3 4 5 6 7 8
```