MULTI-THREADED COLLATZ STOPPING TIME GENERATOR

OVERVIEW

This assignment will explore the use of threads in a computational setting and attempt to quantify their usefulness. It will also introduce a common problem encountered in multi-threaded and multi-process environments: a race condition. You will need to solve the concurrency problem as part of this assignment. However, you should experiment with the application of thread synchronization to prevent race conditions and the bottlenecks it introduces.

THE PROGRAM

When completed, your program (named `mt-collatz`) should produce a list of Collatz sequences for numbers between 1 and N. The program will have two command-line arguments. The first argument, N, defines the range of numbers for which a Collatz sequence must be computed. The second argument, T, is the number of threads your program must create and launch to compute the Collatz sequences for the given range in parallel. For example, the following execution would use 8 threads to compute Collatz sequences for numbers between 1 and 10,000:

```
./mt-collatz 10000 8
```

Each thread computes a Collatz sequence according to the following formula:

$$a_i = \begin{cases} n, & i = 0 \\ f(a_{i-1}), & i > 0 \end{cases}$$

$$f(n) = \begin{cases} \frac{n}{2}, & n \text{ is even} \\ 3n + 1, & n \text{ is odd} \end{cases}$$

The smallest value of *i* for which $a_i = 1$ is called the stopping time. More details on the problem can be found here. The thread counts the stopping time for each Collatz sequence in a global array so that all threads together compute a histogram of Collatz stopping times across a range of numbers. A visualization of a histogram of Collatz stopping times can be found here. Stopping times are shown on the X-axis, frequency values are on the y-axis. Your program must initialize the array with zeros before it creates the threads and computes the Collatz stopping times for numbers 1 through N. Each thread selects a number in the specified range, computes the stopping time and records the result in the global histogram array before it chooses the next number not yet selected by another thread. This process continues until all threads together have computed stopping times up to and including the value of *N*. The final histogram should be printed to standard out (`stdout`) in the following format:

0,*frequency_of_stopping_time(0)*
1,*frequency_of_stopping_time(1)*
2,*frequency_of_stopping_time(2)*
…

Each line contains a value *k* in the range of 0 to 1000 and the corresponding frequency of Collatz stopping times for *k*, i. e. Collatz numbers that stop at the value k (a_k=1). For example, for k=0, the frequencies of stopping time 0 will be 1 because only one number namely 1 will produce a stopping time of 0.

Your program must measure the runtime required to produce the entire histogram. The runtime must be measured using appropriate library calls. It must be written to the standard error stream (`stderr`) before the program terminates. The format of the timing output should be a comma-separated record containing the values *N*, *T*, and the time required to complete the program in seconds and nanoseconds. The following example would be a possible result of the above execution. Note, the timing values are fabricated and just an example of format.

```
500,8,3.852953000
```

IMPLEMENTATION DETAILS

In addition to the global histogram array, I suggest keeping a global variable called *COUNTER* that represents the next number for which a Collatz stopping times must be computed. *COUNTER* should start at 1 and end at *N*. As each thread becomes ready for work, it performs the following steps repeatedly:

1. increment the value of *COUNTER*,
2. performs the job of computing a Collatz stopping time,
3. record the stopping time in the histogram array,
4. go back to step 1. unless *COUNTER* is greater than the value N.

Because multiple threads may read the value of *COUNTER* simultaneously, there is the possibility of a race condition. Therefore, you must introduce thread synchronization using a mutex variable (also known as a lock) to avoid duplication of values in the histogram array. Once threads reach the highest number to test, they must terminate. All worker threads must join the main thread before the program computes the elapsed time, prints it, prints out the histogram array and finally terminates.

When using the BASH shell, you have the option of redirecting standard input, standard output, and/or standard error to a file. The < and > symbols are used to redirect the first two. To redirect `stderr`, you can use `2>`. For example, this command would display `stdout` on the screen but send `stderr` to the file *results.csv*.

```
./mt-collatz 10000 8 [-nolock] 2> results.csv
```

You can also append to an output file by using the `>>` and `2>>` symbols. The parameter `-nolock` is optional indicating that when it is set, you should not use a lock in the program to prevent race conditions on the *COUNTER* variable. The option will allow you to experiment with race conditions in the code.

Experiment with this functionality before trying to save the output of your program executions. It is easy to delete your output files if the redirection command is not entered correctly. Also, experiment with the appropriate system call for time measurement (see below) to compute the elapsed time from the start of the program to the full completion. The elapsed time should approximate the runtime of your program as assessed by considering the system-wide real-time clock.

SUGGESTED LIBRARY CALLS

Most implementation details are left for you to decide. Below is a list of library calls that you will need to use in the implementation of your code.

For implementing and controlling threads use:

- standard C++ thread library

- standard C++ library for mutex locks

For measuring elapsed time use the function below or an equivalent function:

- `clock_gettime(3)` – get the time of a specified clock

Hint: Use the real-time setting of `clock_gettime()`. Also, this function returns elapsed time measured in seconds and nanoseconds. The nanoseconds count the fraction of a second of the system clock measured at any given time. To compute the correct elapsed time, you may have to "borrow" a second from the difference of seconds to compute the correct time difference measured in nanoseconds.

EXPERIMENTS

Your program should allow you to perform an experiment involving thread counts from 1 to 8 and a large *N* for the range of Collatz stopping times to be computed. Select a reasonably large value of *N* to force your program in taking measurable time to execute. Create a graph to show the time required for the experiment as the number of threads increases for a fixed *N*. Analyze the results and discuss them in a report (see below). Keep in mind that the runtime will be significantly influenced by the number of threads you use to compute a solution and the use of locks to prevent race conditions. As time measurements are influenced by other programs running on your machine, you must run your time measurement tests at least 10 times for a fixed T and N and take the average of each run.

DELIVERABLES

Your project submission should follow the instructions below. Any submissions that do not follow the stated requirements will not be graded.

1. Follow the submission requirements of your instructor as published in *Canvas* under the first module.
2. You should have at a minimum the following files for this assignment:
    a. `mt-collatz.cpp`
    b. report.pdf (the report of your experiment)
    c. histogram of your Collatz stopping times for a chosen value of N given in an Excel spreadsheet
    d. chart of measured times in an Excel spreadsheet
    e. `Makefile`

You may combine the histogram (c) and chart (d) in a single Excel spreadsheet and use tabs to separate the two data sets and visualizations. Include both histogram and the graph from your runtime experiment in your report. Submit a *README* file that describes any significant problems you had. If you had no problems, then say so. Keep in mind that documentation of source code is an essential part of programming. If you do not include comments in your source code, points will be deducted. If you do not refactor code appropriately, points will be deducted. Finally, submit a report that includes the following:

- a description of your experiment
    - include how and on which machine you conducted the experiment; describe the machine if possible, including the number of CPUs and cores and memory available
- a visualized histogram of your Collatz stopping time
- the results from your time experiments, including the graph that visualizes number of threads vs. runtime with and without the use of locks
- your analysis on the performance improvements through parallel execution
- your analysis on the impact of the use of locks on performance

- your conclusions from the experiment

TESTING & EVALUATION

Your program will be evaluated on the department's public Linux servers according to the steps shown below. Notice that warnings and errors are not permitted and will make grading quick!

1. Program compilation with Makefile. The options *–g* and *–Wall* must be enabled in the Makefile. See the sample Makefile that I uploaded in *Canvas*.
    - If errors occur during compilation, there will be a substantial deduction. The instructor will not fix your code to get it to compile.
    - If warnings occur during compilation, there will be a deduction. The instructor will test your code.
2. Perform several evaluations with input of the grader's own choosing. At a minimum, the test runs will address the following questions.
    - Are Collatz stopping times computed for numbers in a given range?
    - Are multiple threads used to compute the Collatz stopping times?
    - Does the use of multiple threads affect the compute time of the program?
    - Does the program prevent race conditions, if required?
    - Does the use of locks affect the compute time of the program?

DUE DATE

The project is due as indicated in the schedule in the syllabus and calendar in *Canvas*. Upload your complete solution to the dropbox. I will not accept submissions emailed to me or the grader. Upload ahead of time, as last-minute uploads may fail.

GRADING

This project is worth 100 points total. The rubric used for grading is included below. Keep in mind that there will be deductions if your code does not compile, has memory leaks, or is otherwise, poorly documented or organized. The points will be given based on the following criteria:

Correct Submission Format	**Perfect**	**Deficient**		
Canvas	5 points individual files have been uploaded	0 points files are missing		
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	5 points make file works; includes clean rule	3 points missing clean rule	2 points missing rules; doesn't compile project	0 points make file is missing
compilation	10 points no errors, no warnings	7 points some warnings	3 points many warnings	0 points errors
Documentation & Program Structure	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	10 points follows documentation and code structure guidelines	7 points follows mostly documentation and code structure guidelines; minor deviations	3 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor;

				review sample code and guidelines
Collatz & Threads	**Perfect**	**Good**	**Attempted**	**Deficient**
creates threads with C++ thread library	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
computes Collatz sequence for a number	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
computes stopping times concurrently	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
main thread joins with created threads	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
computes elapsed time using clock_gettime or equivalent	5 points correct, completed	3 points minor errors	2 points incomplete	0 points missing or does not compile
addresses race-condition problems via locks	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile
Chart, analysis, and conclusions	**Perfect**	**Good**	**Attempted**	**Deficient**
histogram and chart are submitted	5 points provided	3 points minor errors	2 points incomplete	0 points no histogram or graph
report with conclusions	10 points a detailed report was provided	7 points there are minor issues with the content provided or the organizing of the report (e.g., missing headings)	3 points the report is mostly incomplete; details about the experiment and your analysis and conclusions are missing	0 points no report

I will evaluate your solution as attempted or insufficient if your code does not compile. This means if you submit your solution according to my instructions, document and structure your code properly, provide a makefile, submit the requested reports and charts, but the submit code does not compile or crashes immediately, you can expect at most 40 out of 100 points. So be sure your code compiles and executes.

COMMENTS

I strongly recommend you start to work on this project right away to leave enough time to handle unexpected problems. Use gdb and valgrind to check for any issues with your code.