Attack of the Threads

Due Wednesday, April 29 at 8 a.m.

CSE 1325 - Spring 2020 - Homework #10 - 1

Assignment Overview

Running multiple threads in a single program is increasingly necessary to take full advantage of the multi-core hardware common in today's market. Let's put those cores to work, and explore how well they perform by solving a polynomial for its roots!

(If you skipped algebra or have just forgotten, a root is an x value for which f(x) is zero. You can read more about polynomials at https://en.wikipedia.org/wiki/Polynomial.)

Because this assignment posted several hours late, you have an extra day to complete it.

Full Credit

Get the Baseline

In your Ubuntu Linux 18.04 git-managed cse1325-prof clone, `git pull` (or clone again) and then copy the **cse1325-prof/P10/full_credit** directory to your own **cse1325/P10/full_credit** directory, using the file manager or `cp -r`.

Add, commit, and push. This will be your starting point for this assignment.

Test the Baseline

First, verify that the code builds and runs by typing `make` (or, if your machine is multi-core, `make -j4`). This will build 4 executables:

- `./test` is the usual regression test. If it prints nothing, all is well. If you get errors, send them to me for advice. (The main program is test.cpp.)
- `./poly` is a menu-driven non-threaded program for creating, evaluating, and solving polynomials. It allows you to enter polynomials, save them to a file, reload them from a file, evaluate them for a given value of x, and solve for the zeros with perhaps the least efficient algorithm possible (we have to solve them slowly enough to tell the difference when threads enable our program to run faster!). The video will walk you through this. (The main program is main.cpp.)
- `./polyt` is the threaded version of poly - or it will be, *once you add threading!* For now, it's identical to `poly`. (The main program is also main.cpp.)
- `./polyb` is a non-interactive ("batch") program that loads and solves the polynomial in the file untitled.poly using the number of threads specified on the command line (default is 1). So, `./polyb 12` will (once you've added threading) use 12 threads to find the zeroes of the polynomial in untitled.poly. We'll use this program with the bash `time` command to determine how much faster your threaded solution is compared to the non-threaded solution. (The main program is batch.cpp.)

```
ricegf@pluto:~/dev/cpp/202001/P10/full_credit$ make clean
rm -f *.o *.gch ~* a.out poly polyt polyb test
ricegf@pluto:~/dev/cpp/202001/P10/full_credit$ make -j4
g++ --std=c++17 -c main.cpp -o main.o
g++ --std=c++17 -c polynomial.cpp -o polynomial.o
g++ --std=c++17 -c term.cpp -o term.o
g++ --std=c++17 -c batch.cpp -o batch.o
g++ --std=c++17 -c -pthread polynomial_threaded.cpp -o polynomial_threaded.o
g++ --std=c++17 -c test.cpp -o test.o
g++ --std=c++17 main.o polynomial.o term.o -o poly
g++ --std=c++17 -pthread batch.o polynomial_threaded.o term.o -o polyb
g++ --std=c++17 -pthread main.o polynomial_threaded.o term.o -o polyt
g++ --std=c++17 test.o polynomial.o term.o -o test
ricegf@pluto:~/dev/cpp/202001/P10/full_credit$ ./polyt
Loaded untitled.poly: +x^4-26x^2+25


   Polynomial Paradise
   ========== ========

1) Load or Enter Polynomial
2) Save Polynomial
3) Evaluate Polynomial
4) Solve Polynomial
0) Exit

==> 4
Solve between min max (== exits): -10 10
Number of threads: 12
Elapsed time: 6 seconds

x =
      -4.9999999000000006 f(x) =   -0.0000239999985752
      -0.9999999670175701 f(x) =    0.0000015831566138
       0.9999999000266961 f(x) =    0.0000047987183827
       5.0000000320859996 f(x) =    0.0000077006400261
      -4.9999999693942989 f(x) =   -0.0000073453682035


   Polynomial Paradise
   ========== ========

1) Load or Enter Polynomial
2) Save Polynomial
3) Evaluate Polynomial
4) Solve Polynomial
0) Exit

==>
```

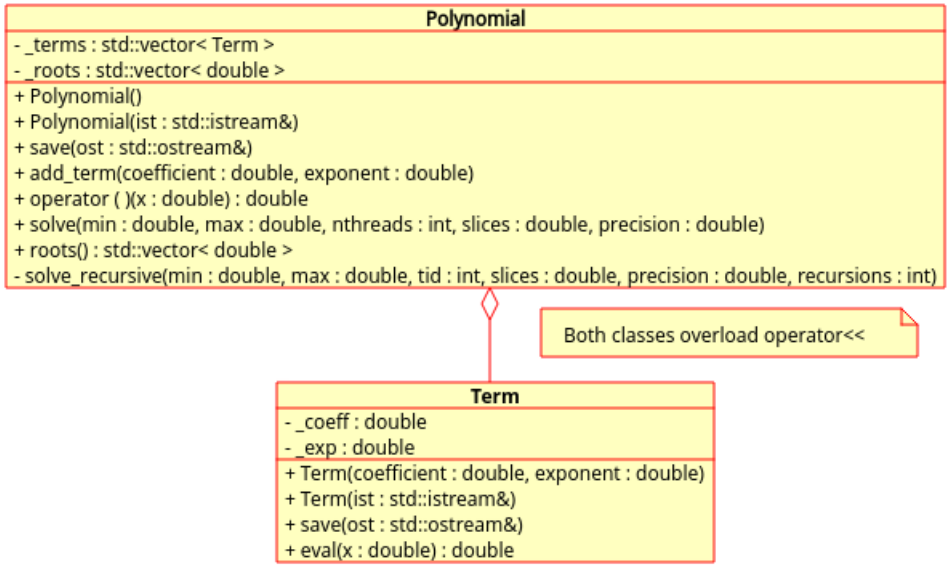
Calibrate the Code Run Time to your Machine

Second, edit **cse1325/P10/full_credit/polynomial_threaded.h** to adjust how long polyt or polyb takes to solve the polynomial in untitled.cpp. **Adjust the default value of the Polynomial::solve parameter named slices** - larger numbers cause the algorithm to run longer by searching more diligently for solutions, while smaller values require fewer calculations.

You can also try other polynomials if you like - all of the files with .poly extension are polynomials with non-imaginary roots close to 0, or you can create your own with `./poly`. More complex and higher order polynomials generally take longer to solve than simple polynomials.

Add Threading to polyt

Third, edit **cse1325/P10/full_credit/polynomial_threaded.cpp** to create a threaded version of this program. The Polynomial class models a polynomial as a vector of Term objects, as the Term class represents something like $3x^2$ (where 3 is the coefficient and 2 is the exponent).



Guidance for adding threads to the Polynomial class is below.

You can test your program by running either `./polyt` for the menu version (you'll be prompted for the number of threads to create when you ask it to solve the polynomial), or `./polyb` (you can specify the number of threads to use on the command line).

`./poly` will continue to use the unthreaded version for comparison.

Evaluate Threading Performance

Once your program is threaded and runs reliably, **complete results.txt that you copied to your full_credit directory from cse1325-prof.** This will guide you through testing and recording the performance improvements you may expect from threaded applications.

And, as always, add, commit, and push all files.

```
ricegf@pluto:~/dev/cpp/202001/P10/full_credit$ time ./polyb 12
Requested 12 threads to solve +x^4-26x^2+25 -- 4 roots found

real    0m6.407s
user    0m23.504s
sys     0m0.020s
ricegf@pluto:~/dev/cpp/202001/P10/full_credit$
```

Bonus

In for a pence, in for a pound. For bonus credit, copy your full_credit solution to the bonus directory, and then copy main.cpp to maint.cpp, and adjust your Makefile to use maint.cpp when building polyt.

Now, modify maint.cpp to port loading *multiple* polynomials at the same time (I'm thinking a vector of Polynomials here). For option 4, instance one thread per polynomial running Polynomial::solve.

So, if you've loaded 3 polynomials, selecting option 4 would create 3 threads, each running Polynomial::solve - and each of those threads would (of course) instance multiple additional threads to search for solutions based on each nthreads parameter.

You do **not** need to complete a results.txt file for the Bonus portion. The code's the thing, with apologies to The Bard.

Add, commit, and push all files.

Extreme Bonus

The provided code for Term's operator<< overload prints the polynomials to the console using the carat and normal exponent, e.g., 5.2x^3.14. Fix this by printing properly superscripted exponents, e.g., $5.2x^{3.14}$.

The exponents (which are doubles, and thus may include +, -, ., and e) must be displayed using Unicode superscript characters. See https://en.wikipedia.org/wiki/Unicode_subscripts_and_superscripts.

The Extreme Bonus has nothing to do with threads. Rather, it will give you some exposure to the fun (and I use that term lightly) that is the Unicode standard and its support (and I use that term *quite* lightly) in C++.

```
==> 1
Filename (blank to enter by hand): poly2x.poly
Loaded poly2x.poly: +x³·¹²-5.85988x¹·⁸+8.53975
```

```
ricegf@pluto:~/dev/cpp/202001/P10/extreme_bonus$ make clean
rm -f *.o *.gch ~* a.out poly polyt polyb test
ricegf@pluto:~/dev/cpp/202001/P10/extreme_bonus$ make -j4
g++ --std=c++17 -c main.cpp -o main.o
g++ --std=c++17 -c polynomial.cpp -o polynomial.o
g++ --std=c++17 -c term.cpp -o term.o
g++ --std=c++17 -c batch.cpp -o batch.o
g++ --std=c++17 -c -pthread polynomial_threaded.cpp -o polynomial_threaded.o
g++ --std=c++17 -c test.cpp -o test.o
g++ --std=c++17 main.o polynomial.o term.o -o poly
g++ --std=c++17 -pthread batch.o polynomial_threaded.o term.o -o polyb
g++ --std=c++17 -pthread main.o polynomial_threaded.o term.o -o polyt
g++ --std=c++17 test.o polynomial.o term.o -o test
ricegf@pluto:~/dev/cpp/202001/P10/extreme_bonus$ time ./polyb 1
Requested 1 threads to solve +x⁴-26x²+25 -- 4 roots found

real    0m19.356s
user    0m19.343s
sys     0m0.004s
ricegf@pluto:~/dev/cpp/202001/P10/extreme_bonus$ time ./polyb 12
Requested 12 threads to solve +x⁴-26x²+25 -- 4 roots found

real    0m6.034s
user    0m23.654s
sys     0m0.008s
ricegf@pluto:~/dev/cpp/202001/P10/extreme_bonus$
```

Guidance for Full Credit

Preparation

For this assignment, a working single-threaded solution is provided. Class Polynomial represents a polynomial equation as a series of Term objects, and includes an operator()(double x) overload to evaluate the polynomial at x - that is, for Polynomial f, we can evaluate that polynomial by coding simply f(x). (NOTE: This is one of C++'s strengths relative to Java, in my opinion.)

Several main functions are included that you may use to evaluate this class. You may time the solution on your machine with e.g., 12 threads using `time ./polyb 12`.

Ensure that the polynomial with which you are working can be solved with the provided Polynomial class in no less than 30 seconds on your machine. (NOTE: This isn't an efficient algorithm, but it has the virtue of being easy to break into threads and thus well-suited to this assignment.) You can adjust the solution time by changing the number of X "slices" searched - more slices consume more time and / or by creating a more complex polynomial. Thirty seconds or more will give you enough time to observe the effect (hopefully, the improvement) of threading on solution time.

Goal

Your goal is to split up the search for solutions among an arbitrary number of threads. You should *abstract away* the use of threads, which means the interface to class Polynomial should not change. Thus, the only file you actually need to change to add threads is polynomial_threaded.cpp. When you are done, calling Polynomial::solve with nthreads > 1 would result in that number of threads sharing the work.

(NOTE: Properly, we would want the mutual exclusion object - the mutex - to be a private attribute of the Polynomial class. Because of a design flaw in C++, private attributes are declared with the public interface in polynomial.h.

Although it is not strictly object-oriented, you can avoid modifying polynomial.h by defining the mutex in the global scope of polynomial.cpp. Since we never include .cpp files in other files, this mutex is still only visible within the body of class Polynomial, just as if it were an attribute. We call this "file scope" or more formally "translation unit scope". It works, and we won't deduct points if you'd like to use it here.)

Implementation

You will, of course, need to include the thread library to create a thread and the mutex library to define a mutex.

The remaining changes must be made to the solve method, with a slight change to solve_recursive. In solve, use a for loop to create nthreads threads, each searching a different range of x to search for solutions to the polynomial.

The solve_recursive method is your thread. Note that std::vector::push_back is NOT thread-safe, so ensure that two threads do not try to push simultaneously (a "thread collision").

Evaluation

Once you have the program using threads without segfaults, exceptions, or deadlocks, complete the Threading Evaluation form results.txt for your code. This will guide you through measuring the incremental performance boost of adding threads. You'll commit and push results.txt in your full_credit directory along with your source code.