Parallel Prime Generator

Programming Assignment 1 – COP 6616

Background

This class is my first endeavor into multicore programming having no prior experience in it in any language. So when the opportunity presented I chose to learn a new language that is so well regarded that specializes in this: Rust.

Rust is a very interesting language, very different from others I've programmed in before, I definitely felt like there was a bigger learning curve in getting started with this language as opposed to others.

The numbers shown in this report are representative of what I obtained with my own PC. It is an AMD Ryzen 7 3700x (8 cores/16 virtual), with 32 Gb of RAM running on Manjaro Linux.

The Prime Algorithm

Initially I had chosen a simple Sieve of Eratosthenes in order to cull the ranges of numbers. This proved to be somewhat inefficient, specially considering this specific use case of parallelizing the algorithm. This algorithm doesn't benefit from not starting the search from zero.

I definitely struggled for a bit in how I would be able to not start the algorithm at zero. I was even initializing a 108 array for every thread but just doing the numbers in the given range for that thread.

Then I came upon this parallel prime algorithm:
https://create.stephan-brumme.com/eratosthenes/#blockwise

In this site the author goes into detail going from the most naive implementation to a proper parallel solution. In the end the author adapts the Erastosthenes algorithm to work in a single block/range of numbers by maintaining an array the size of the range but only the odd numbers. This saves spaces and iterations on even numbers.

According to their numbers the algorithm is able to calculate the sum of 109 primes with 8 threads in 0.65 seconds! While their algorithm doesn't need to return the last 10 highest values like this one, it is incredibly quick.

I adapted the prime generation algorithm to work for our use case in the assignment. Although I had trouble with getting the prime number from the range sized

array that each thread generates, I was able to figure out how the algorithm fundamentally works.

The Parallelization

The one thing I noted from the author's algorithm was the they were plainly dividing the whole range of numbers evenly by 8. As we had discussed in class this approach is naive in such that it ignores how the primes are distributed throughout the given range.

Considering this I took a different approach to the problem; rather than have one thread have the highest numbers by itself, I divide them by GROUP_DIVISIONS (a positive integer that is divisible by 2 and greater than or equal to NTHREADS).

This is where the function generate_ranges comes in. It'll divide the whole range in chunks of UPPER_LIMIT/GROUP_DIVISIONS. Since the resulting ranges are sorted in an array, each thread will grap one range from the head and then the tail, and so forth until there are no more ranges up for grabs.

The idea here is the first half of ranges should be easier than the second half of ranges since they're much smaller numbers. So each thread can have a sort of even spread of ranges both from small to high.

Another thing to note is that I did not use a lock or mutex implementation in this assignment, this prevents the program from stalling waiting on data. Rather the set of ranges is assigned per thread and then through a functionality called channels in Rust I pass the data on to the main thread. The idea is of multiple producers, single consumers. While the other threads are working, the main one can be processing the primes as they come out to sort, add and count. While I was unsuccessful in getting the main thread to continue working until all the threads stop, the possibility is there, it is just my inexperience in the language. Hopefully I will eventually be able to make this fix as it allows some sequential bits of logic to be done in parallel.

Once the threads are done, their generated data is put into an array, sorted and then returns the top ten elements.

The Results

This yielded interesting results as I'll go into now. Essentially there are three main variables in this program, number of threads (NTHREADS), the number of group ranges (GROUP_DIVISIONS) and the max number of primes (UPPER_LIMIT). Given that NTHREADS and UPPER_LIMIT are static per the description of assignment, this gave the opportunity to try the efficiency of the algorithm given a varying number of ranges per thread.

The idea is to find the sweet spot between dividing the ranges (so there is less opportunity for a single thread to not be working while the others are) and not having too much overhead in orchestrating the divisions and the number of iterations the sieve algorithm does.

Do note that there are two time variables to consider here: the time it took for all the threads to run (which is the one being measured in the assignment) and the total runtime (which would include the sequential only bits.

Group Divisions	Parralel Runtime (sec)	Total Runtime (sec)
2	0.47653824	0.683
4	0.7574315	1.086
8	0.7041865	0.973
16	0.3544903	0.717
32	0.3770866	0.755
64	0.31082976	0.632
128	0.30818978	0.668
256	0.35709944	0.725
512	0.38944897	0.759
1024	0.3865615	0.745
2048	0.38295773	0.733

Below you can see the data graphed, group divisions is in the x-axis while parralel runtime is the y-axis. Dividing the data up in different number of groups starts off as being slower than just running it all sequentially in one thread but then then delegating the ranges to different threads helps runtime. Eventually this becomes a game of diminishing returns. The fastest execution was with 128 group divisions, which after that number it only starts increasing again the runtime where the overhead might be causing way more cycles.