Final Parallel Project

Jonathan Macoco

December 16, 2022

Problem 1

For my parallel project, I decided to find the factors of a large semi-prime number. This is the basic idea for cryptography strength used in the encryption and decryption of keys. The challenge is to find the original large primes used to produce the semi-prime. I am using prime density analysis and sieves algorithm to generate a list of prime numbers. With this list, I can find any semi-prime number given, with the only constraint being memory. This program speeds up the ability to find these two prime numbers that make up the semi-prime, as when the semi-prime number grows larger, it can take a long time to find the two prime factors sequentially.

The value of this solution and a real-world application would be trying to decrypt RSA keys as they are generated based on the product of two very large prime numbers. This program would be very useful in not just generating the keys but also decrypting the keys.

To start the program, I use Sieve's algorithm to generate all the prime numbers up to the semi-prime for which I am trying the prime factors. After this prime numbers list was generated, I first attempted to find the prime factors by multiplying two numbers from the prime list and checking if the product was equal to the semi-prime. This itself takes a very long time, so I came up with another solution that makes the sequential version of the program slightly faster. In this solution, I iterate through the list of prime numbers and divide the semi-prime whose factors I am trying to find by the prime number in the list. If that number is divisible by the semi-prime, then I check if the result of the division is a prime number. If it is, then I have found the two prime factors. So I am using prime number searching and testing as my numerical method.

To make my program run parallel I used OpenMp as my parallel programming method.

Problem 2

To demonstrate the sequential version of the program I will be using 3 semi-prime numbers and timing them. I timed the program using POSIX clock_gettime functions. The three numbers I will be using are 49, 1385252357, 3537146813, and 4297863209 . Here are the results of my solution.

Semi-Prime	Time in Seconds	Prime Factor 1	Prime Factor 2
49	0.000002	7	7
1385252357	1.159279	86627	15991
3537146813	2.809085	68207	51859
4297863209	3.400566	332987	12907

I was able to verify my results by using google to check if the numbers were prime. I then used desmos to check if the two prime numbers product was equal to the semi-prime.

Problem 3

To demonstrate the parallel version of the code an speed up I will be using the same numbers as I did in the sequential versions test. For the program I am still using using POSIX clock_gettime functions to time the program. I am only time the portion that I have made parallel to get a more accurate time.

Semi-Prime	Time in Seconds	Prime Factor 1	Prime Factor 2
49	0.001387	7	7
1385252357	0.674432	86627	15991
3537146813	1.634642	68207	51859
4297863209	2.032810	332987	12907

As you can see based on the results I do get speed up but not for numbers that are really small as this violates Gustafson's law.

Semi-Prime	Speedup in Seconds
1385252357	0.484847
3537146813	1.174443
4297863209	1.367756

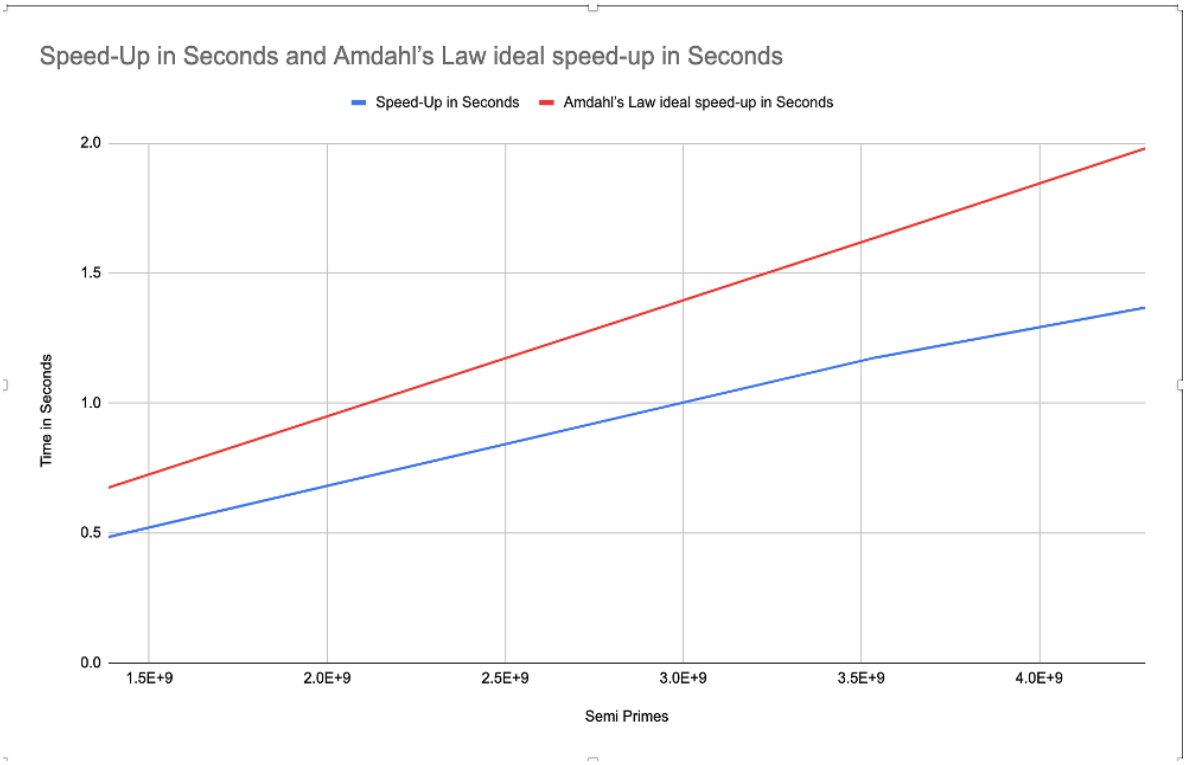
Problem 4

Here is the table for calculating the speed up, parallel and sequential portions.

Amdahl's Law Parameter	How obtained	Description
Sequential portion (% of total): %44.2932176339	$p - 1$ where p is equal to 0.557067823661. So the sequential portion equals $0.557067823661 - 1 =$ .442932176339	The only part of the program that runs sequentially is attaining a prime numbers list using seives algorithm.
Parallel portion (% of total): %55.7067823661	$p = \frac{s(1-\frac{1}{su})}{(s-1)}$ where su is speed up and s is number of threads. So when I input my values I get $\frac{4(1-\frac{1}{1.71762537008})}{(4-1)} =$ 0.557067823661	The part of my program that runs in parallel is searching for the two prime numbers that make up the semi-prime.
Number of shared memory cores used and type	I used 4 shared memory cores.	The type of parallelism was instruction level parallelism as I use a for directive
Final value used for S, the scaling factor	The value I used for S was 4	I used 4 as the value for S since there are 4 cores on the school's machine.

Using Amdahl's law I was able to calculate the ideal speed up. $\frac{1}{(1-0.557067823661)+\frac{0.557067823661}{4}} =$ 1.71762537008. So the ideal speed up should be about 1.72x faster than the sequential version. Below are the results

Semi-Prime	Actual Speed Up	Ideal Speed Up
1385252357	0.484847	0.674931227841
3537146813	1.174443	1.63544684943
4297863209	1.367756	1.97980657437

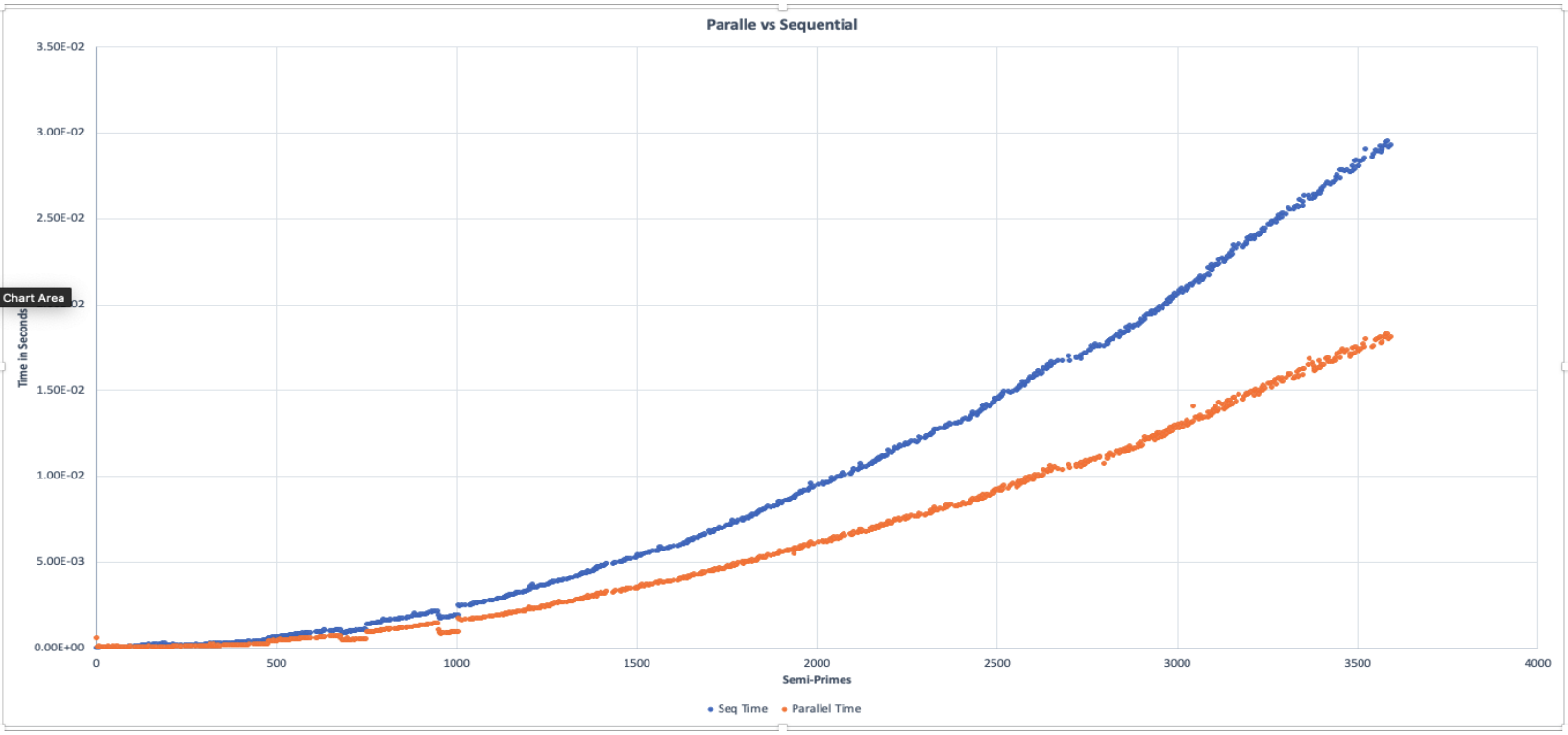


Comparison between ideal speed up and actual speed up

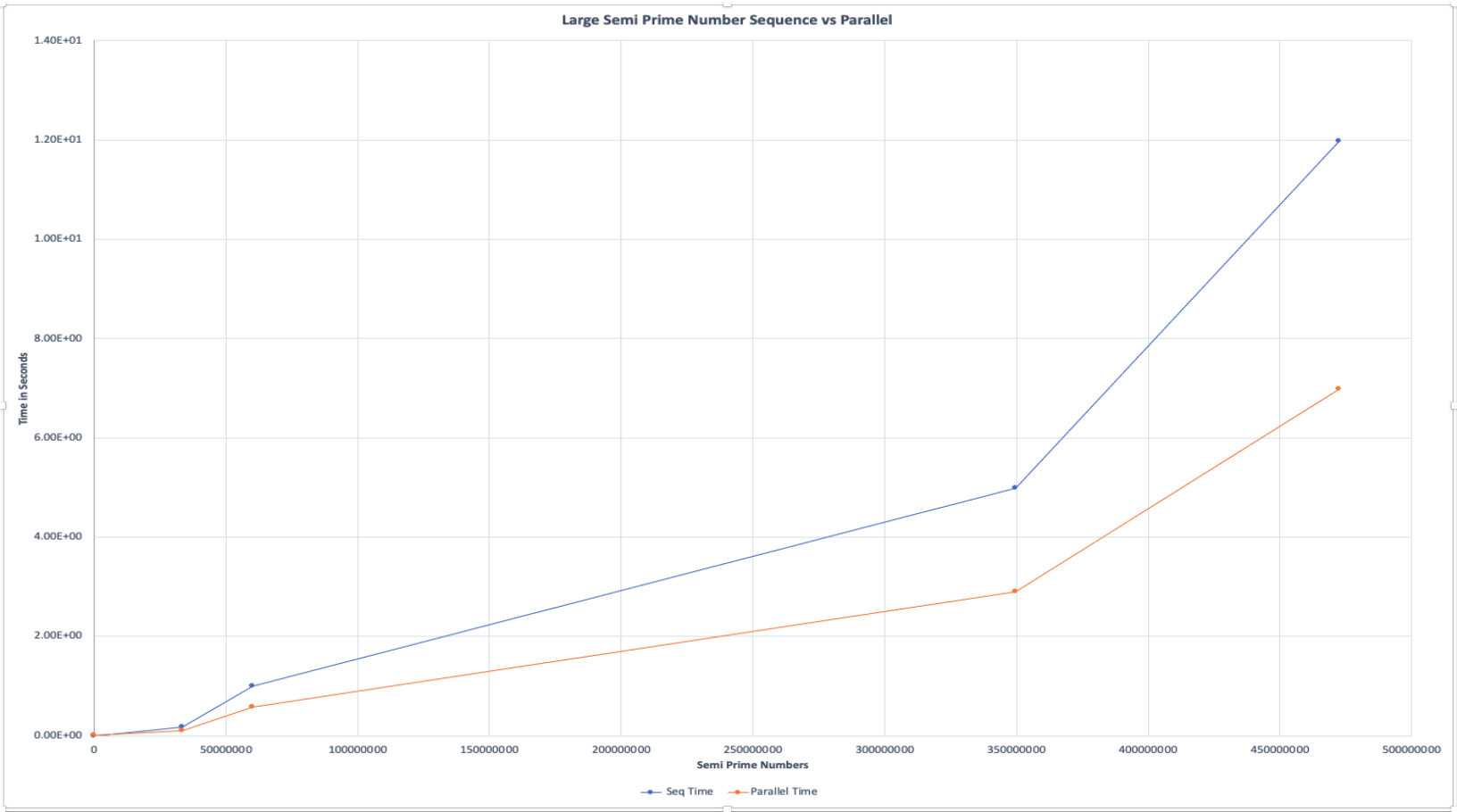
So based off the output I don't get as much speed up as Ahmdal's ideal speedup result. But I do get speed up nonetheless.

Appendix

To further demonstrate the speed-up and how this program can affect large semi-primes, I wrote another program that calls the same algorithm for different-sized semi-primes in a given list. This allowed me to automate testing for a large set of semi-prime numbers. In doing this, I could see which semi-prime numbers benefit most from parallelism. Using this large set of semi-prime numbers, I can show the benefit of my parallel program for finding two prime factors of a semi-prime. The code's output is in CSV format, allowing me to generate graphs to show this. I made two graphs, one in ascending order and one with five very large random semi-prime numbers. The ascending order graph shows that with small semi-primes, there isn't much speed up, but as the semi-primes increase, the more effective the parallel version of the program is. I used the five very large random semi-prime numbers to show how big the numbers can get and how much faster the parallel version of the program is when used with large numbers. To get these times, I used POSIX clock gettime functions.



Comparison between parallel version and sequential version for semi-primes in ascending order



Comparison between parallel version and sequential version for large semi-primes