CS475 PA2

By Bruce Darcy

In this project, I improved worked with a prime number generation algorithm, over 4 steps of improvement.

Sieve1 improved the programs complexity and did various changes to make it run faster sequentially.

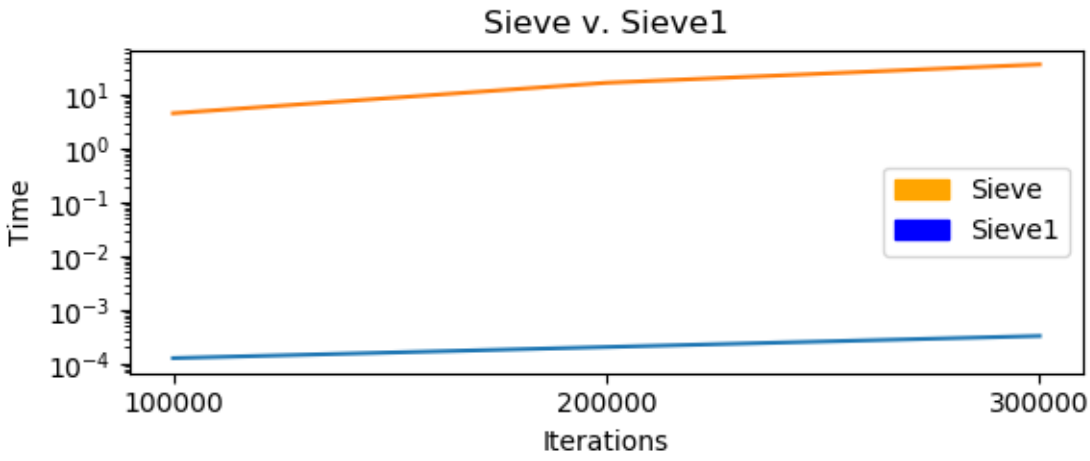
Sieve2 parallelized sieve1 to check mark off multiples of different primes concurrently.

Sieve3 implemented blocking to improve caching of the program.

Sieve4 parallelized sieve3 by running the process of marking off non-primes concurrently by block.

Sieve1

Sieve1 has 4 optimizations to the algorithm over sieve. Sieve1 moves over values by a jump by multiple of primes, marks off only the multiples of a prime less than the square of the prime, and therefore only checks multiples of primes that are less than sqrt(n). It also only stores and checks odd numbers. Here both functions are graphed with time on a logarithmic scale to show how much sieve1 improved on the original algorithm. The complexity of sieve is actually greater than sieve1, since in sieve1 we don't check for multiples of primes over sqrt(n).

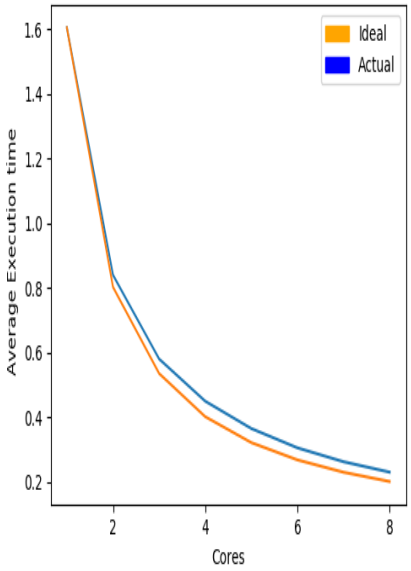


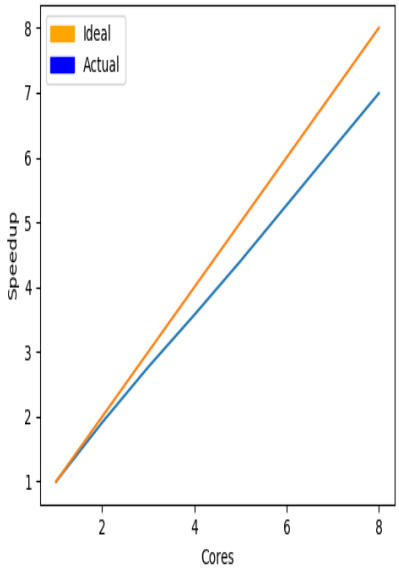
sieve1	sieve3
4.5591325	0.00012675000000000002
16.8272015	0.000204
36.9269935	0.0003275

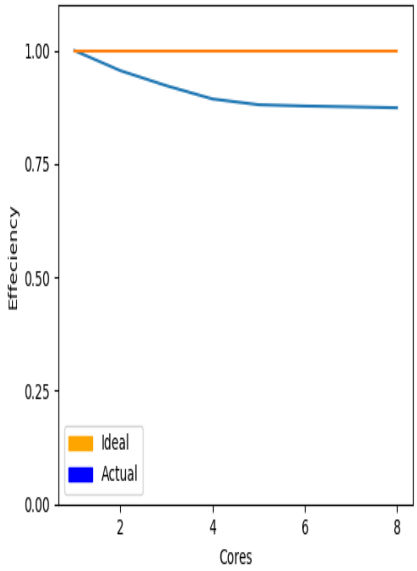
Sieve2

Sieve2 parallelizes the inner for loop of sieve1. Below are the graphs for 500 million, 1 billion, and 1.5 billion in descending order. Initially with 500 million iterations, the program gets pretty good speedup, however in the 1 billion and 1.5 billion cases, the program becomes bottle-necked by memory access latency and because cache misses start to become more common, and the speedup suffers as a result.

N=500 million





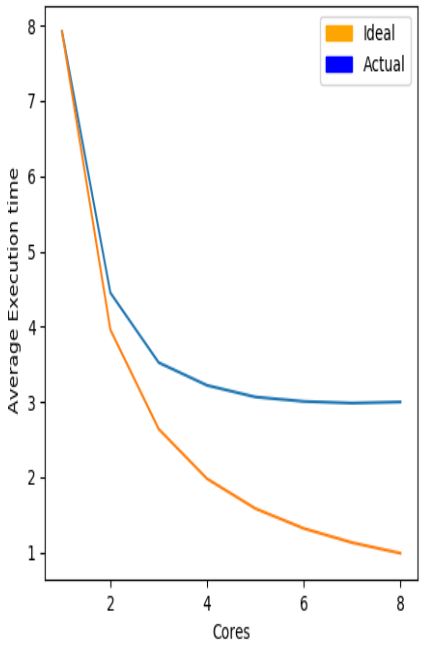


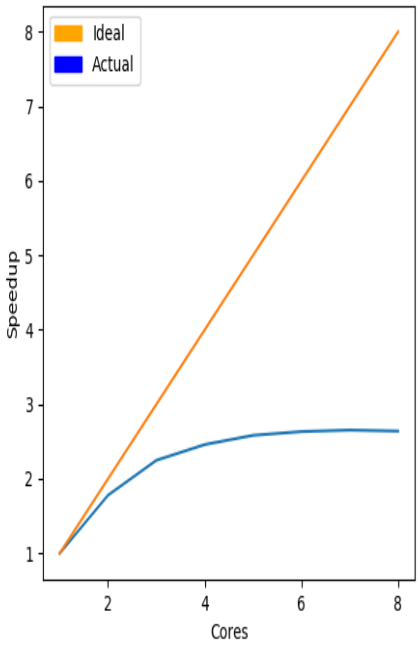
1	1.6032942857142858
2	0.8388808571428571
3	0.5794180000000001
4	0.4487697142857427
5	0.3641972857142857
6	0.30449071428571434
7	0.2615274285714285
8	0.22936085714285714

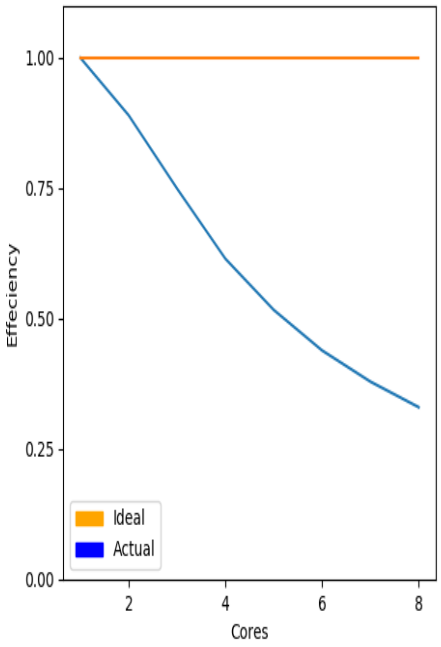
1	1.0
2	1.911230029941252
3	2.767077111367416
4	3.5726436848934293
5	4.402268629129973
6	5.2654948426764125
7	6.130501471574685
8	6.990269855486614

1	1.0
2	0.955615014970626
3	0.922359037122472
4	0.8931609212233573
5	0.8804537258259947
6	0.8775824737794021
7	0.8757859245106693
8	0.8737837319358267

N= 1 billion





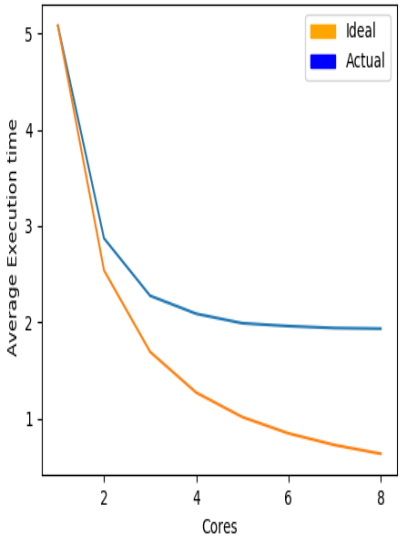


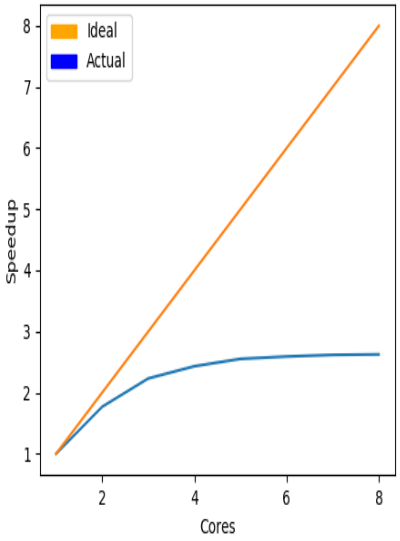
1	7.918465285714285
2	4.450312571428571
3	3.522693428571429
4	3.220582285714286
5	3.0657867142857143
6	3.0072164285714287
7	2.984610428571429
8	2.999713571428571 4

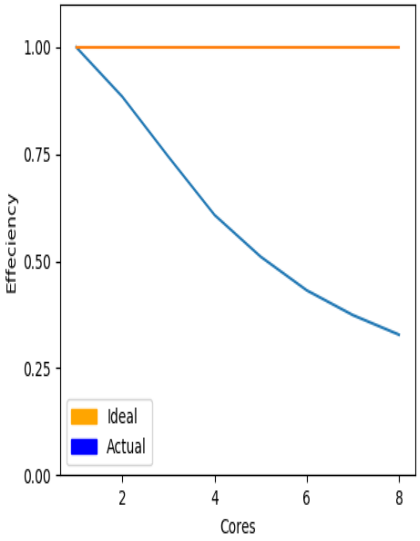
1	1.0
2	1.7793054215004094
3	2.2478439995629964
4	2.4587060920128194
5	2.5828493707068523
6	2.6331544382643366
7	2.65309844457805
8	2.6397404609344846

1	1.0
2	0.8896527107502047
3	0.7492813331876654
4	0.6146765230032049
5	0.5165698741413705
6	0.4388590730440561
7	0.37901406351114997
8	0.32996755761681057

N=1.5 billion







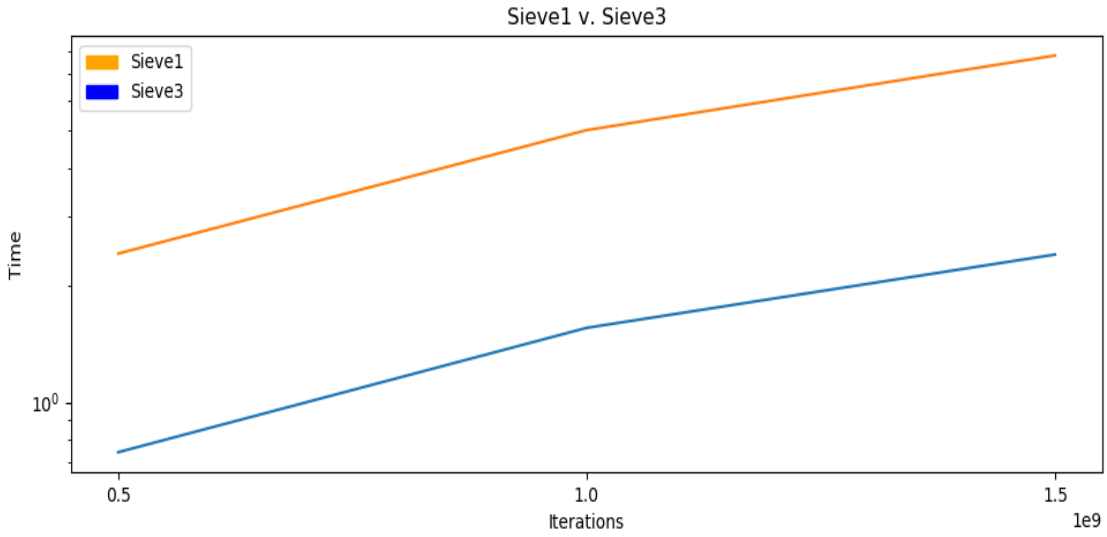
1	5.079253857142858
2	2.8719680000000003
3	2.2757530000000004
4	2.0888924285714285
5	1.990289142857143
6	1.960651285714286
7	1.9406035714285716
8	1.934538

1	1.0
2	1.7685621347949758
3	2.2319003236040365
4	2.4315535772306407
5	2.5520180700234225
6	2.5905952242254195
7	2.6173577808082538
8	2.6255642727839192

1	1.0
2	0.8842810673974879
3	0.7439667745346789
4	0.6078883943076602
5	0.5104036140046845
6	0.4317658707042366
7	0.3739082544011791
8	0.3281955340979899

Sieve3

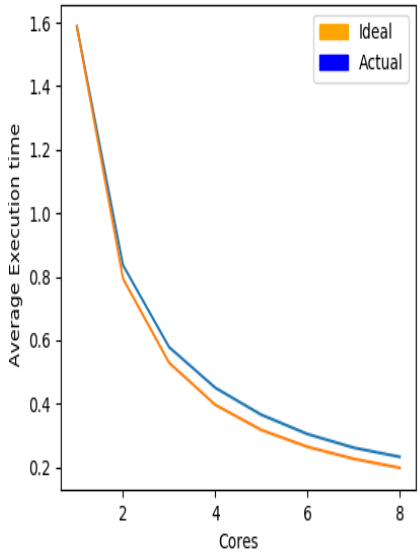
Sieve3 implements blocking to keep portions of numbers in the caches as all prime numbers are checked against them before moving on to the next block. In cachegrind, blocking correctly improved cache miss rate on 2 of the caches from about 50% to 3%. As shown in the graph, sieve3 runs a lot faster than sieve1 because of the avoided cache misses.

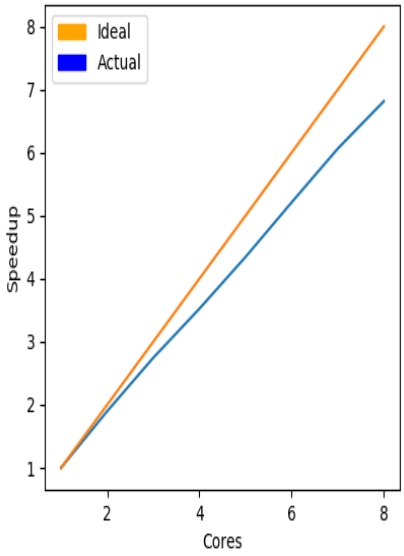


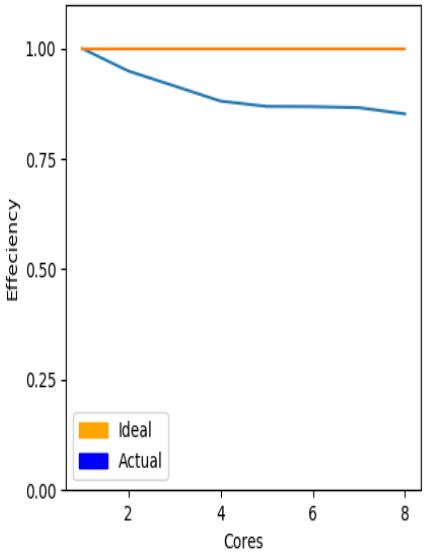
sieve1	sieve3
2.4141975	0.74345625
5.02116925	1.5544259999999999
7.817114	2.40178725

Sieve4

Sieve4 parallelizes sieve3 by parallelizing the blocking loop, allowing the program to mark off multiples of prime numbers in multiple blocks at once. The speedup, execution time, and efficiency are all much closer to ideal than in sieve2 when the program was bottle-necked by cache misses.







1	1.587761
2	0.8363812857142857
3	0.5783042857142858
4	0.4505718571428572
5	0.3654404285714286
6	0.3047257142857142
7	0.261881
8	0.23296714285714284

1	1.0
2	1.8983698309844672
3	2.7455459681383743
4	3.523879653887456
5	4.344787483439747
6	5.210459523318395
7	6.062910253130239
8	6.815385983308499

1	1.0
2	0.9491849154922336
3	0.915181989379458
4	0.880969913471864
5	0.8689574966879494
6	0.8684099205530659
7	0.8661300361614627
8	0.851923247913562 3

Conclusion

The most important part of improving the algorithm was improving its complexity in sieve1. Parallelizing can be bottle-necked and rendered more useless by memorylatency, and it can be improved by blocking data in applicable algorithms to greatly reduce cache misses.