Algorithm Design

Sieve

The provided sieve algorithm is a doubly nested loop. These loops scale with N which causes our time complexity to be O(n^2). The space complexity is O(n) as an array is allocated which scales with our problem size.

Sieve1

Our first round of improvements on the sieve algorithm reduces our complexity significantly. For example, our space complexity is reduced by half, but it still scales with n, so our space complexity is still O(n). Our doubly nested loops now start from sqrt(N) and iterate by 2 after each iteration. However, these improvements do not change how the problem scales with size, so we consider it to still be O(n^2).

Sieve2

Next, we parallelized sieve1. For this, I chose to parallelize the inner loop of our calculation. This loop is safe to parallelize and it performs a large amount of work O(n). We could parallelize the outer loop, but the work needed to fix race conditions is not worth the speedup as there is not as much work done (O(sqrt(n)) compared to the inner for loop (O(n)).

Sieve3

Now, we update the algorithm to utilize locality of data more effectively. To do this, we split our problem size into blocks, and for each block, we mark off multiples of every prime up to sqrt(N). This allows us to precompute the primes. From here, we can find the largest block size to utilize the cache most effectively. This algorithm is O(n^2). One of inner loops don't scale linearly with N. For example, the loop scales with the number of primes up to sqrt(N). So, although the number of primes does increase, it will increase slower than N. This is why I consider this algorithm to be O(n^2) instead of O(n^3). Space complexity is O(n). Slightly larger than previous algorithms due to the new array allocated for precomputing primes.

Sieve4

Finally, we parallelize the algorithm. This allows us to utilize cache effectively and split up the work as each block can be marked off independently from one another. After marking is complete, we return to the master thread to count the number of primes by counting the threads which are not marked.

Experimental Design

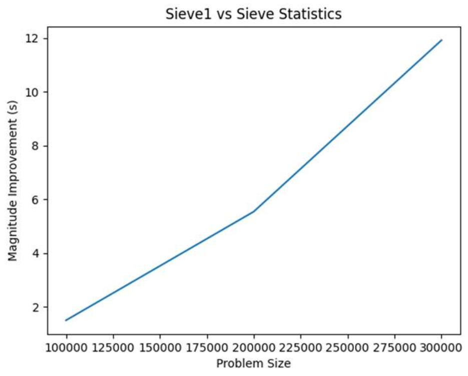
Experimental Conditions

I used the Cheyenne state capital machine for this experiment. This machine has 8 cores. The L1d and L1i cache are 256 KiB, the L2 cache is 2 MiB, and the L3 cache is 20 MiB. I am the only user on this machine

Experimental Results

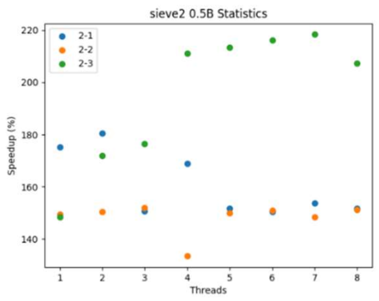
Sieve1

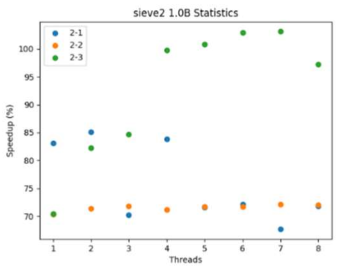
Sieve1 implements several improvements over the baseline sieve algorithm. In the following graph, we can see the magnitude of improvement of sieve1 over sieve. We see a 12x improvement for a problem size of 300,000. Magnitude of improvement is calculated by finding the difference between the execution time and dividing the result by 10.

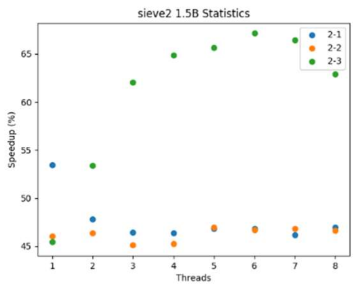


Sieve2

Before experimenting on state capitol machines, I tested locally to determine which parallelization results in the largest speedup. This experiment was conducted with outer for loop parallelization (2-1), outer and inner for loop parallelization (2-2), and inner for loop parallelization (2-3). Experiments were conducted using an N of 0.5 billion, 1.0 billion, and 1.5 billion.



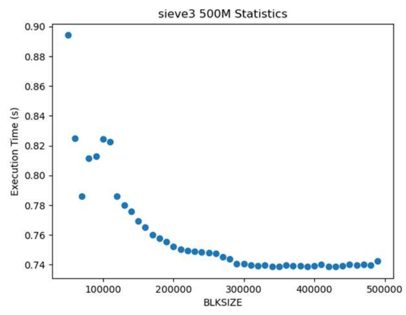


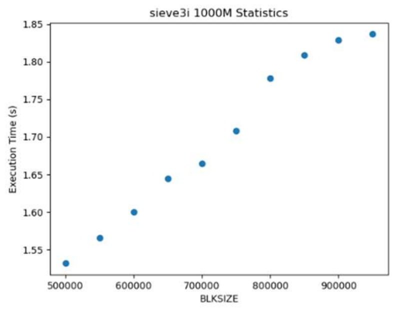


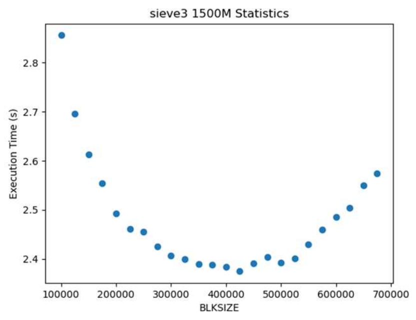
The results are most apparent for N = 1.5B, the inner for loop (2-3) parallelizing the marking of primes performed best. However, we see that outer for loop parallelization and the inner and outer parallelization does not yield consistent speedup. This is likely due to a race condition between threads which are marking primes and reading the next unmarked prime. Additionally, parallelizing the outer for loop does not yield significant performance gains as there is negligible iterations being performed (sqrt(N)) compared to the inner for loop (N). Because of the obvious speedup improvement and lack of race conditions, I decided to proceed with further testing using sieve2-3 which only parallelizes the inner for loop that marks a prime’s multiples.

Sieve3

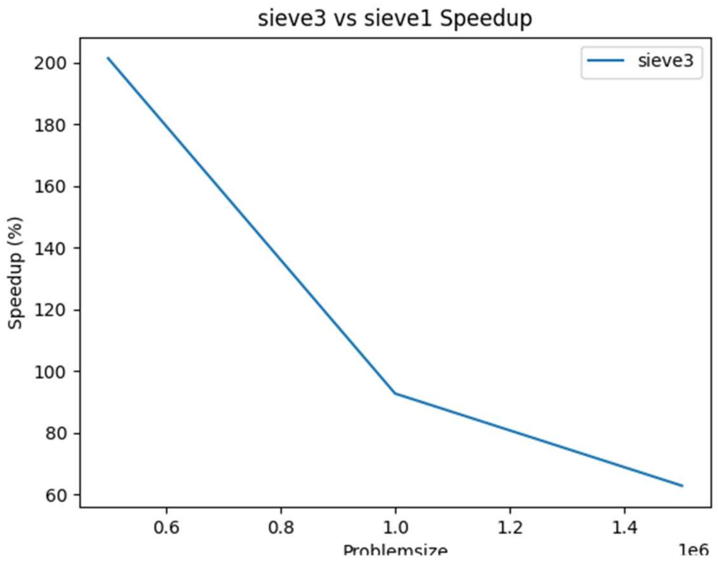
For this experiment, we changed our algorithm to utilize locality of data. This algorithm blocks our problem size into a variable size. To do this we have a triply nested for loop, however, the performance gains from our data being within the CPU's cache should outperform the losses incurred by increased complexity. We begin our experimentation by determining an optimal size for our blocks. I tested these block sizes using problem sizes of 500 million, 1 billion, and 1.5 billion.







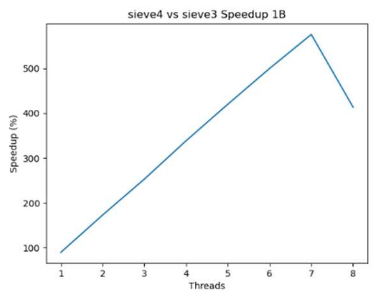
The shape of the graph for each problem size was similar, so above are graphs from each experiment. In the left-most graph, we see the execution time asymptotically approach the optimal value around ~350,000. We begin to see an increase in execution time around 500,000. In the middle graph, I confirm that the execution time increases with block sizes above 500,000. Note, we see that a block size of 1 million is not optimal, execution time has been increasing as we approach a block size of 1 million. Finally, we see the full polynomial curve with its minimum around 400,000. This will be our final block size for further testing.

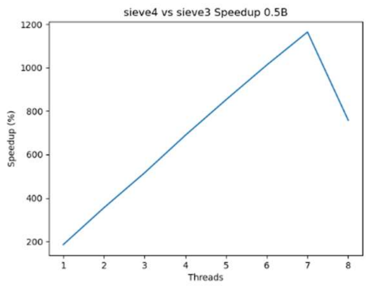


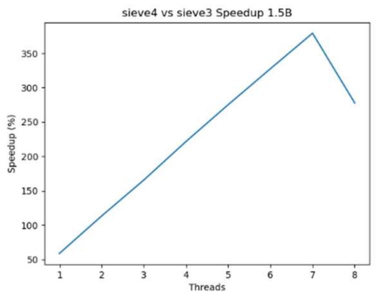
Above is a graph of sieve3's speedup in comparison to sieve1. We see that for small problem sizes, we have a speedup of 200%! Speedup decreases to 80 percent as we hit a problem size of 1.5 billion.

Sieve4

For our loops which mark off multiples of primes, there can be a race condition. For example, if there are two threads within the same block, and one is marking multiples of 3 and another thread is marking multiples of 5, will be marked for both 15. Although they are both setting the value of those indices to 1, I noticed in my testing that there was a slowdown. From my testing, I found the best performance from parallelizing code safe from race conditions.







We see that for large datasets (N = 1.5B) we have a speedup of ~375%! This shows how data locality is often more important than the complexity of an algorithm, given that the startup costs associated with parallelizing the algorithm do not outweigh the benefits of locality. We also see that we have maximum speedup for 7 threads.

Conclusion

After all experiments, we can conclude that improving locality can result in massive performance gains, more so than the gains from just improvements in the algorithm. Additionally, the benefits of locality can be compounded by adding parallelization if the associated overhead costs don’t outweigh the benefits of locality.