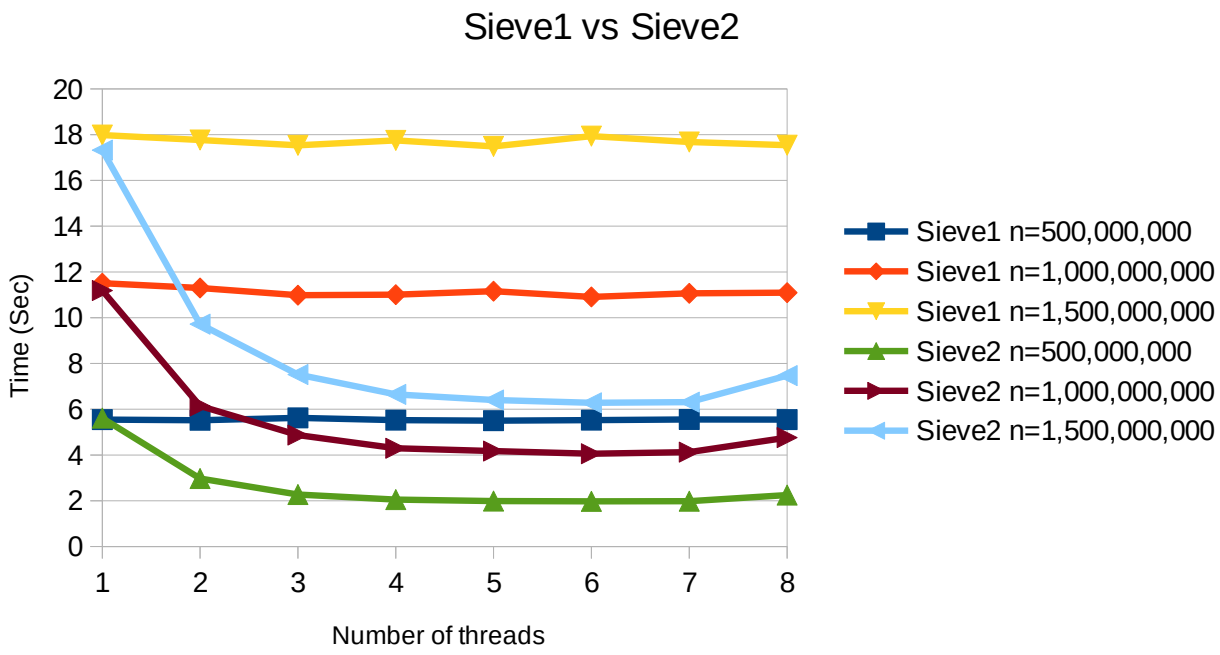
Algorithm Description: The Sieve algorithm is a faster way of calculating prime numbers up to a certain number, and can be improved by giving the program a certain block size used for calculations.

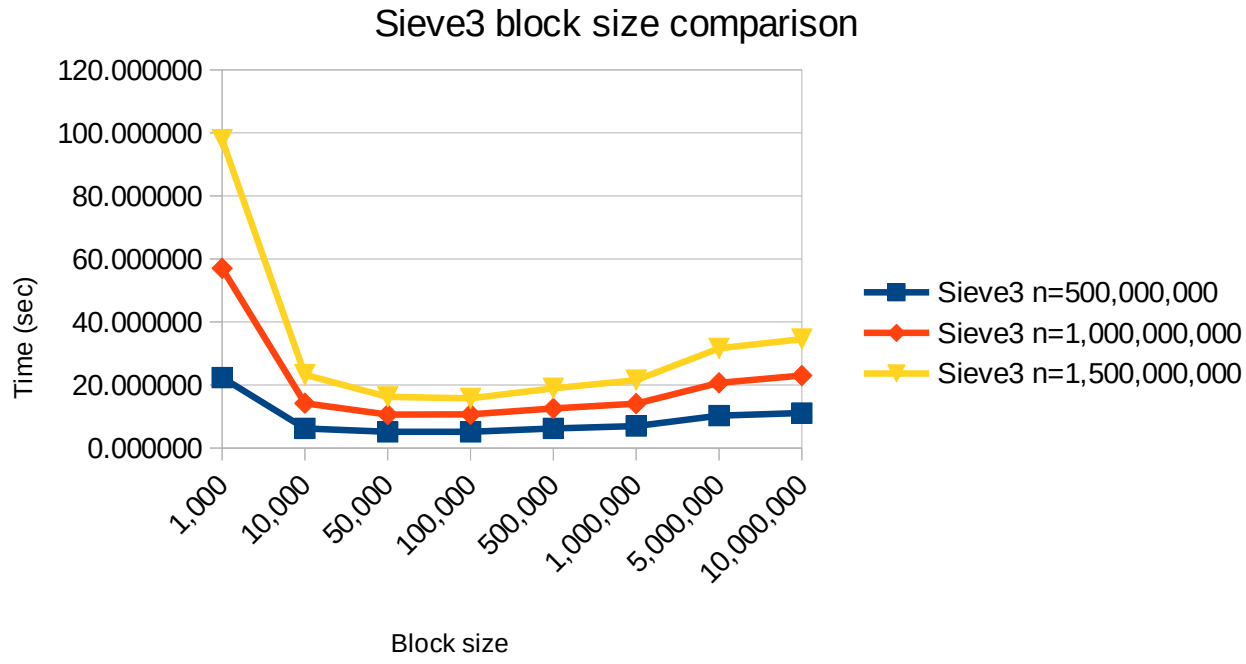
Description of parallel approach: My parallel approaches for both sieve2 and sieve4 were to parallelize all of the "for" loops that take up a significant amount of computation time, but also to make sure they didn't run if the input size, N, was too low. The first "for" loop for both of the programs are for the initiation of the "mark" array, which definitely takes up a good chunk of time and can very easily be parallalized. Sieve4 also has two more "for" loops that are used for more pre-computation, and will be parallelized as well. The second place most of the time is taken up is the computational "for" loops, but this is where sieve2 and sieve4 differ. Sieve2 uses a "while" loop with a nested "for" loop, while sieve4 uses three nested "for" loops that can all be parallelized and not just one. Overall I think sieve4 will be much faster, even though it has more pre-computation it can be better parallelized, and it's just using a better algorithm all-together.

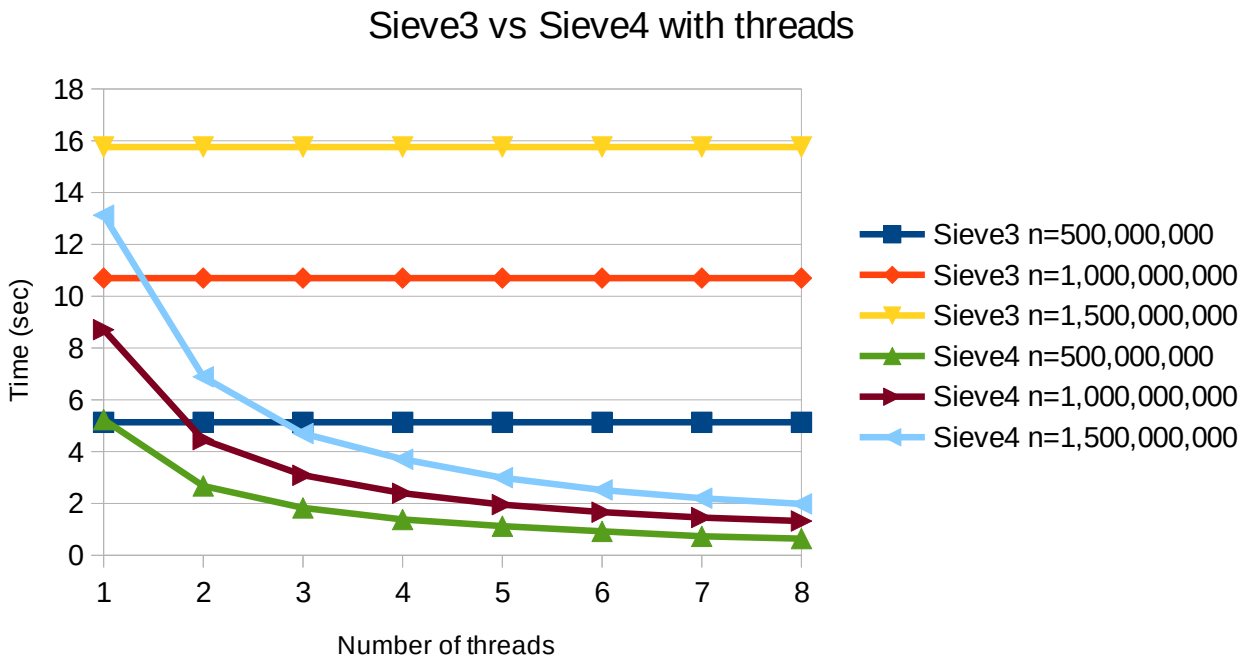
Machine description:

- **Name**: boise
- **Number of cores**: 8
- **Cache sizes**:
  - L1d cache: 32K
  - L1i cache: 32K
  - L2 cache: 256K
  - L3 cache: 15360K

Experimental results:







Conclusion: Parallelizing sieve1 and sieve3 made clear and drastic improvements to speed. Since this time I'm using a machine with 8 cores, I found 8 threads to be the ideal number for most programs. I found 100,000 to be a very good block size, so it's the size I used when comparing Sieve3 vs Sieve4 (figure 3).