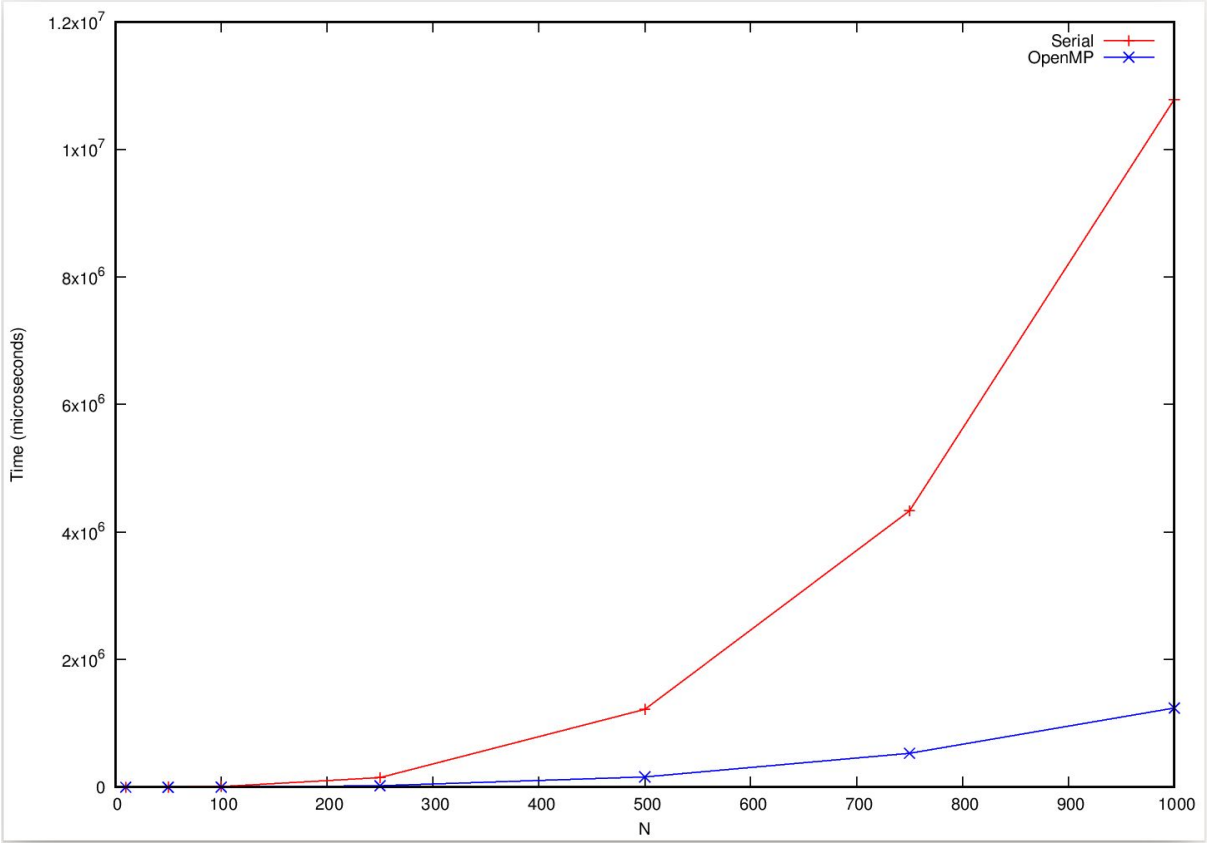
Gaussian elimination results:

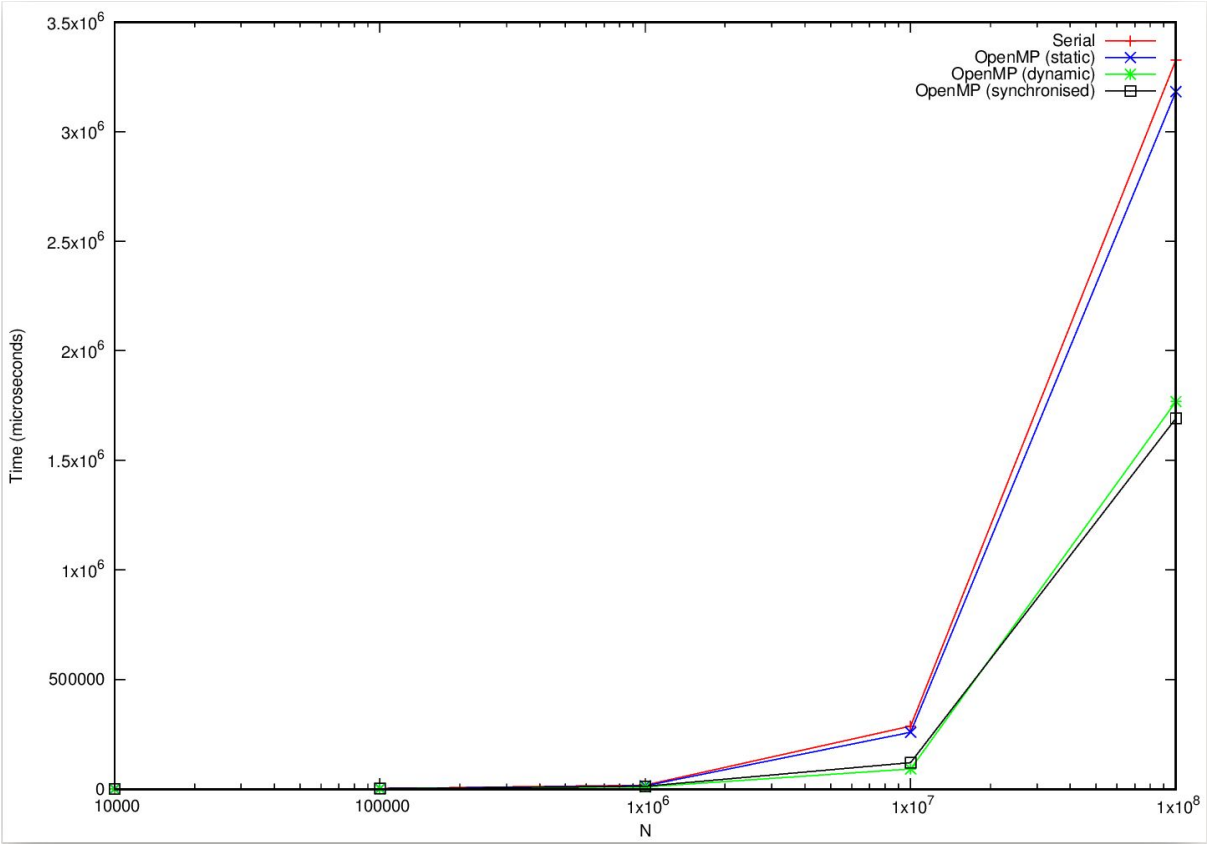


Original file: gauss.eps

The gaussian elimination code benefited greatly from using OpenMP, and it was easy to do so as well. All it required was inserting “#pragma omp parallel for” before three of the for loops.

With *N* = 1’000 and running on a Lonsdale node with 8 cores the OpenMP version was almost 8.5x faster than the serial version. I suspect the speed up being higher than the 8x increase in core usage was because of less demand on each core’s cache. With smaller values of *N* the OpenMP version doesn’t have as high a speed up, but still outperforms the serial version by a large margin.

Sieve of Eratosthenes results:



Original file: sieve.eps (note x-axis uses log scale)

Despite using 8 cores the best speed up I could get in my sieve code was 1.96x with N = $1x10^9$.

My first attempt was to just add "#pragma omp parallel for private(i)" outside the outer for loop. This gave a slight speedup, but not even close to the full theoretical 8x speed up. I then changed the scheduling from static to dynamic, and this resulted in faster code, but again not close to the theoretical 8x speed up. I believe the dynamic scheduling was better as the inside of the loop can take almost no time if *primes[i]* is not prime, or it can take a long time if *primes[i]* is prime and *i* is not close to the limit. This results in an extremely unbalanced workload. By using dynamic scheduling work can be better divided between the cores.

This approach has the downside of duplicate work being done at times as *i* is private. It relies on some threads getting slightly ahead and and marking entries in the *primes* array as not prime, therefore allowing other threads to skip those values of *i*. If the threads all reach the same value of *i* at the same time then duplicate work will be done.

I tried using OpenMP's barrier feature to synchronise the cores and divide work evenly among them while avoiding duplicate work. While this avoids the duplicate work it requires cores to wait for every other core to reach the barrier. This version was just slightly faster than the dynamic version for large values of N.