STUDY OF SCALABILITY

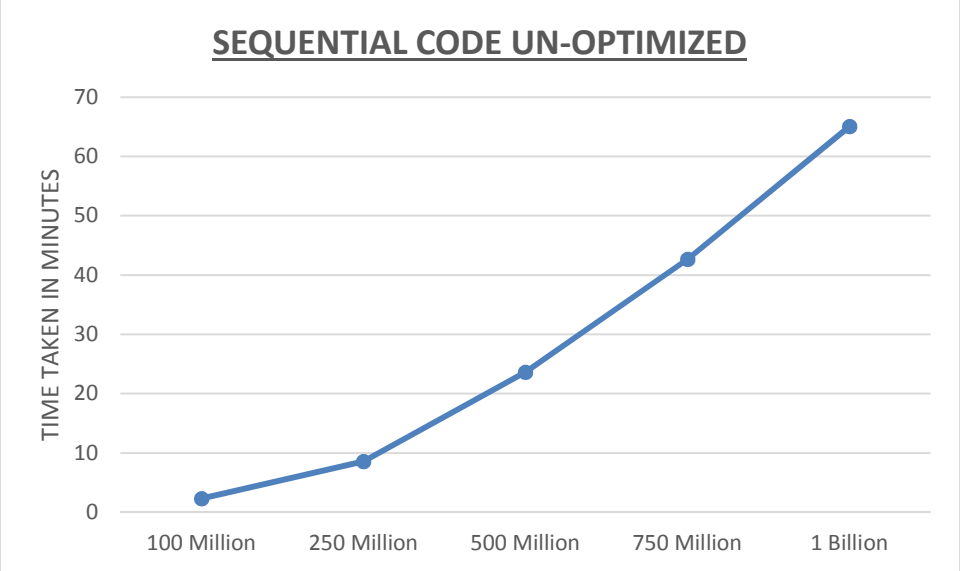
Contents

Rationale for Parallelizing

Time taken by the sequential code to find all prime numbers from 1 to the following problem sizes: 100 Million, 250 Million, 500 Million, 750 Million and 1 Billion.

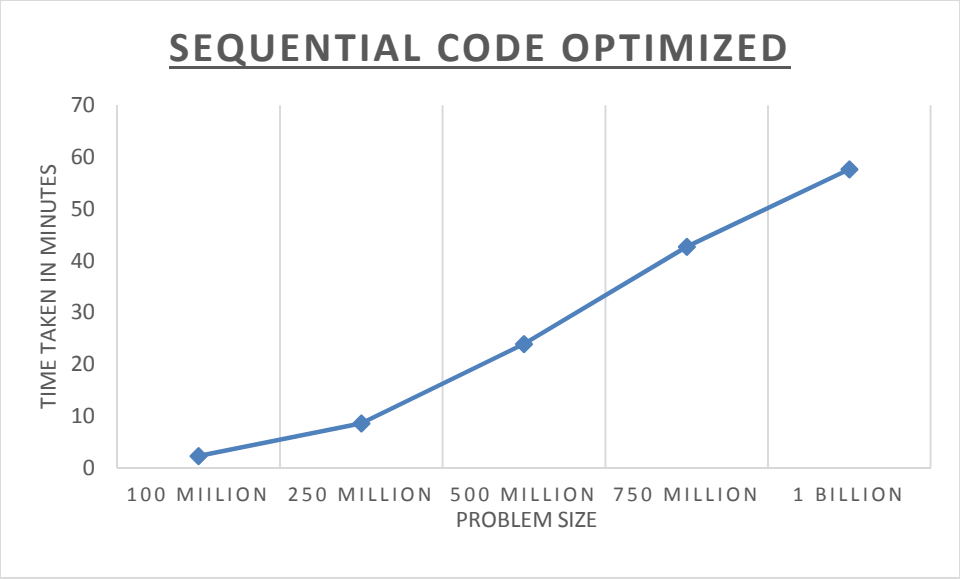
With un-optimized code:

SEQUENTIAL CODE (Un-Optimized)		
	Time in Seconds	Time in Minutes
100 Million	138.228412	2.303806867
250 Million	516.615147	8.61025245
500 Million	1418.3403	23.639005
750 Million	2560.776334	42.67960557
1 Billion	3902.902359	65.04837265



With Optimized Code (-O3):

SEQUENTIAL CODE (Optimized)		
	Time in Seconds	Time in Minutes
100 Million	137.46	2.291
250 Million	516.04	8.600666667
500 Million	1433.85	23.8975
750 Million	2561.81	42.69683333
1 Billion	3460.41	57.6735



The graphs and the numbers clearly illustrate that it takes a huge amount of time to compute the prime numbers and the nature of the algorithm is such that it is fully parallelizable.

The problem does not require movement of data or Inter process communication and therefore is an embarrassingly parallel task. However; given a candidate number the program is required to search till the square root of the given number for a factor; the number of computations that need to be performed increases with larger numbers. Thus load has to be uniformly distributed when the algorithms is parallelized.

Largest Prime Number Generated

Largest Prime number begotten: 999999937 and

# of prime numbers found primes: 50847533 (excludes 2)

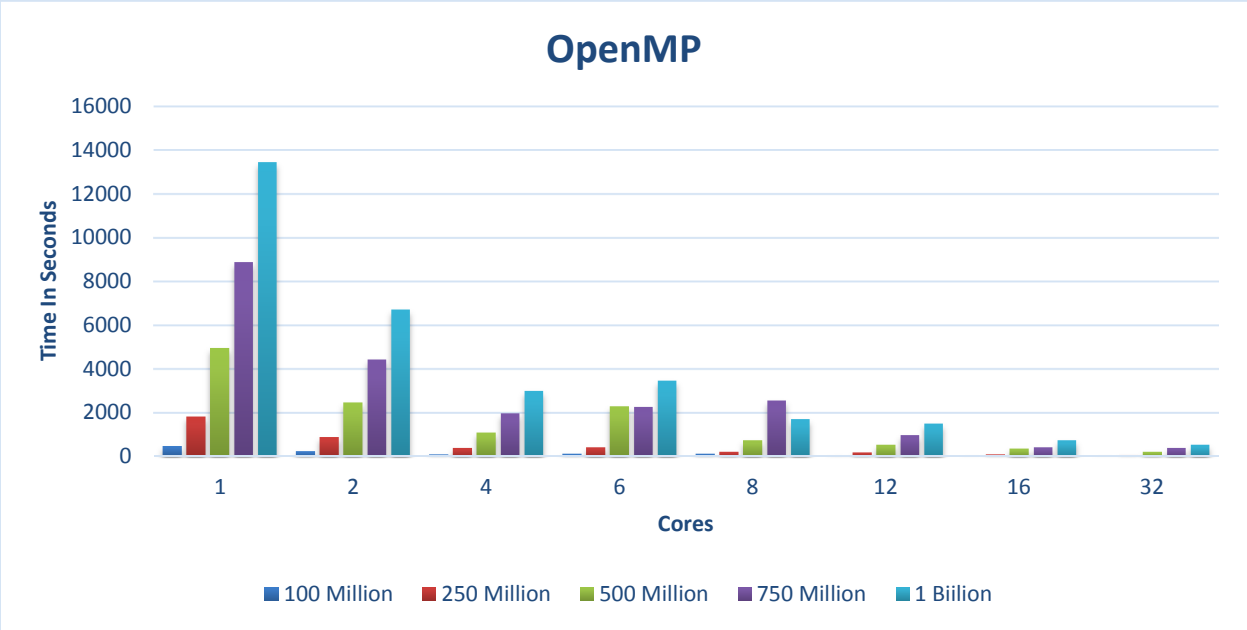
Shared Memory OpenMP

	OpenMP (Optimized with -O3) Time in Milli-seconds							
Problem Size\N Cores	**1**	**2**	**4**	**6**	**8**	**12**	**16**	**32**
100 Million	0.715	4.77	12.124	3.067	7.187	17.485	25.508	79.701
250 Million	0.663	0.927	1.808	3.806	25.104	8.568	31.351	34.775
500 Million	0.682	2.82	1.135	2.866	23.481	15.565	16.001	41.44
750 Million	0.637	1.403	1.692	22.895	13.656	16.821	16.175	35.115
1 Biilion	0.663	2.12	1.399	31.017	14.91	7.505	25.924	43.644

Interesting thing to note here is that the OpenMP optimized code on a single processor beats the sequential times. And even more surprisingly it beats the times recorded when the program is run on

multiple processors. This is quite un-realistic therefore let's take a glance at the performance of the un-optimized code.

	OpenMP (Un-optimized Code) Time taken in Seconds							
Problem Size\ N Cores	**1**	**2**	**4**	**6**	**8**	**12**	**16**	**32**
100 Million	489.753	244.118	108.392	122.254	136.368	54.591	29.906	20.23132
250 Million	1825.153	909.977	404.968	433.467	231.887	203.892	108.005	74.24411
500 Million	4953.069	2475.95	1101.869	2286.631	743.427	554.429	355.997	213.5333
750 Million	8885.803	4436.543	1975.564	2277.571	2573.091	997.895	426.764	407.6689
1 Biilion	13458	6733.828	2993.293	3458.751	1715.813	1515.889	736.0062	540.5702



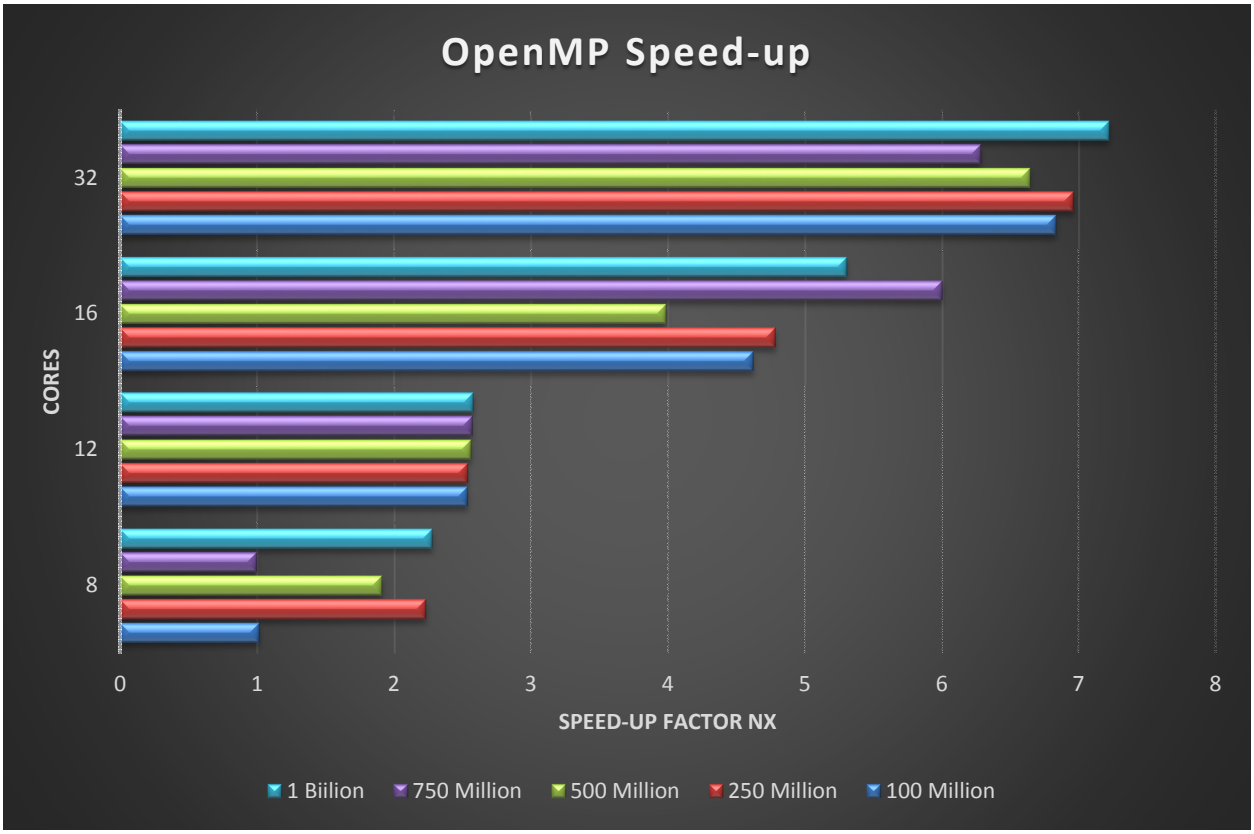
As can be seen from the graph the run times for a given problem size decrease with increasing number of cores.

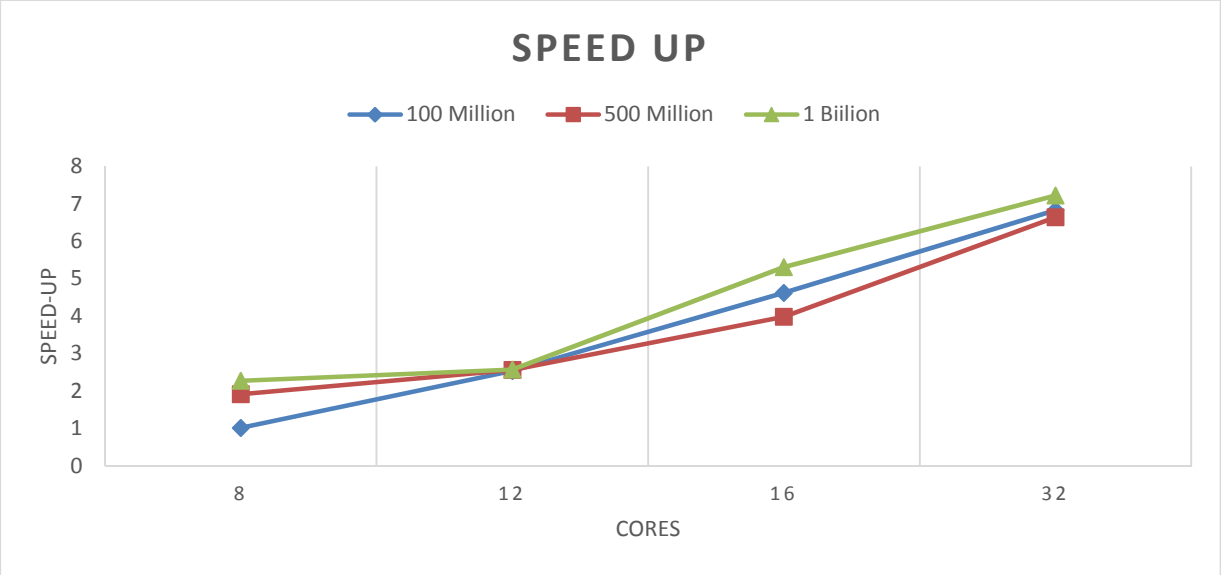
Time taken in minutes for the sake of comparison with sequential code times.

	OpenMP (Un-optimized Code) Time taken in Minutes							
Problem Size\ N Cores	**1**	**2**	**4**	**6**	**8**	**12**	**16**	**32**
100 Million	8.16255	4.068633	1.806533	2.037566667	2.2728	0.90985	0.498433	0.337189
250 Million	30.41922	15.16628	6.749467	7.22445	3.864783333	3.3982	1.800083	1.237402
500 Million	82.55115	41.26583	18.36448	38.11051667	12.39045	9.240483	5.933283	3.558888
750 Million	148.0967	73.94238	32.92607	37.95951667	42.88485	16.63158	7.112733	6.794481
1 Biilion	224.2999	112.2305	49.88822	57.64585	28.59688333	25.26482	12.26677	9.009503

Speed-ups due to OpenMP:

Speed-ups				
Problem Size	**8**	**12**	**16**	**32**
100 Million	1.013643	2.532073	4.622096	6.832398
250 Million	2.227875	2.533769	4.783252	6.958332
500 Million	1.907841	2.5582	3.984136	6.642244
750 Million	0.995214	2.566178	6.000451	6.28151
1 Biilion	2.274667	2.574662	5.302812	7.219973



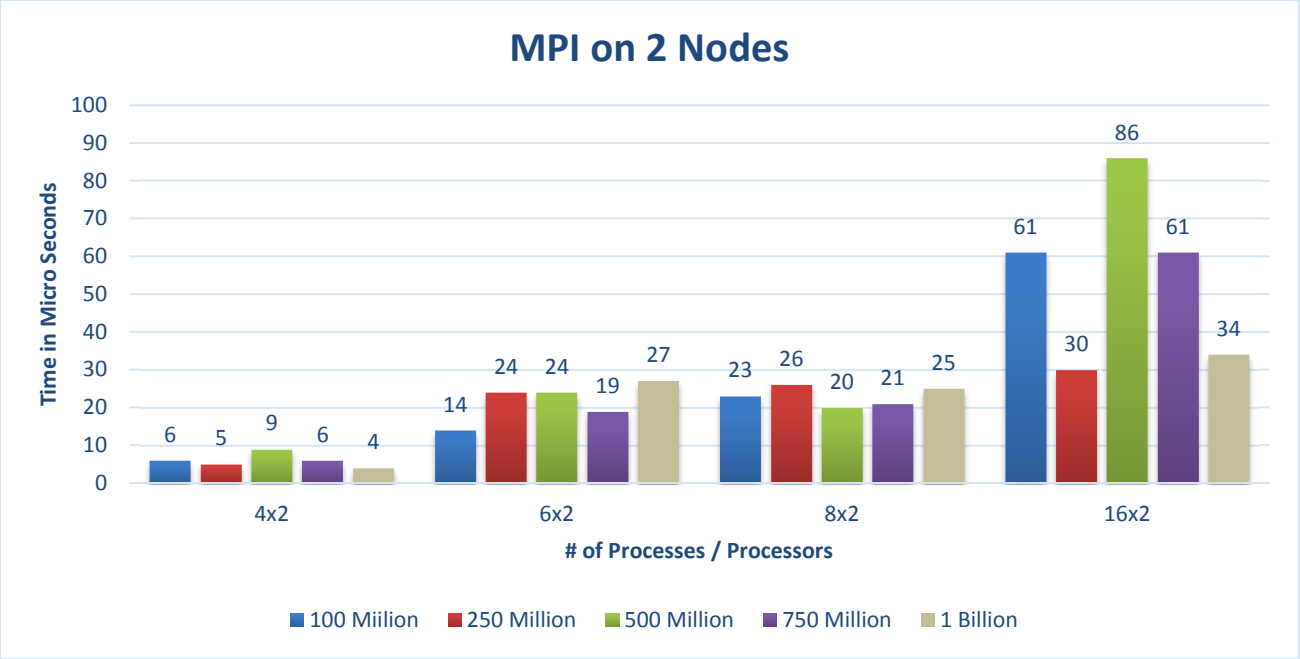


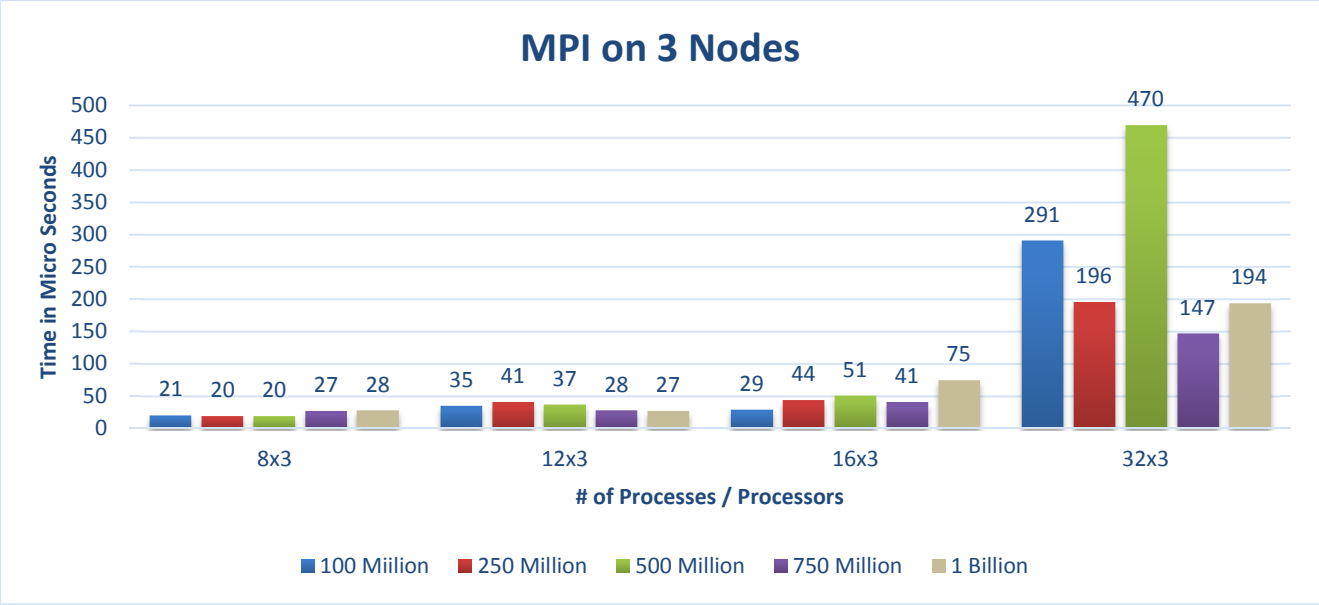
Distributed Memory MESSAGE PASSING INTERFACE

The MPI code was run on 2 nodes and subsequently on 3 nodes for different processor counts as shown below in the table. Following are the times taken: Please note the times are in Micro Seconds.

Note: All print statements associated with the algorithm had been taken out. The profiling was performed on code that does not write to the standard out.

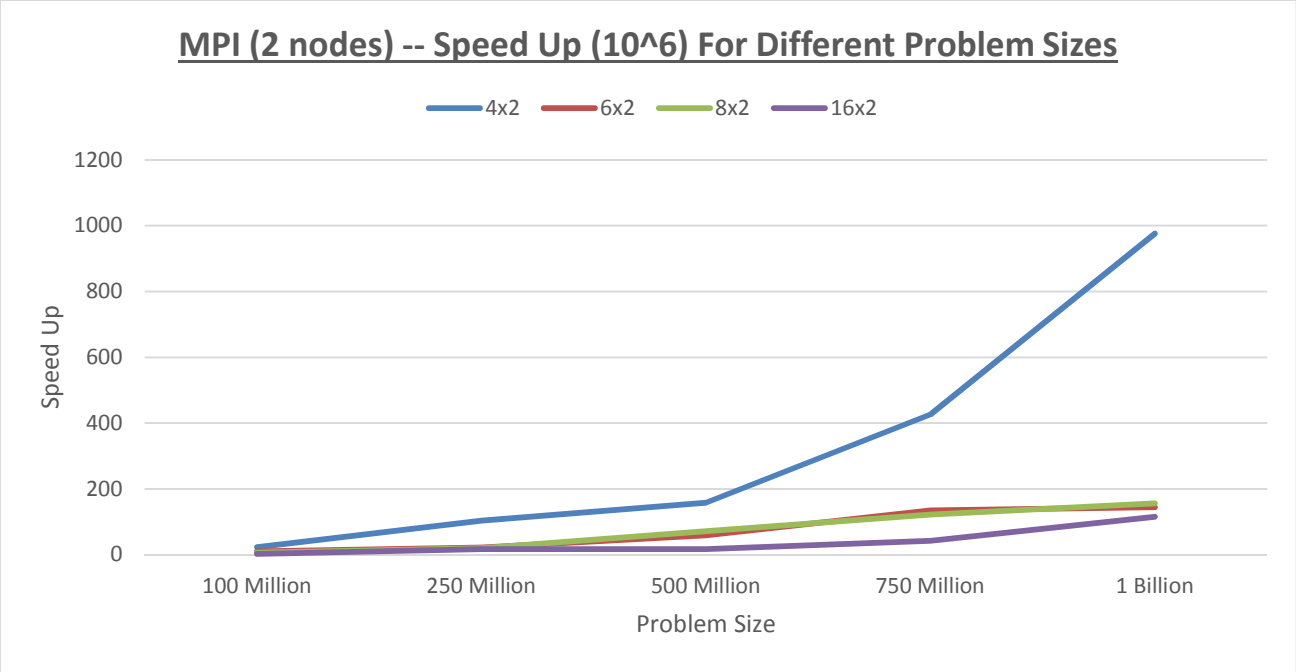
	MPI (# Tasks/Node x # Nodes) Time in Micro seconds							
Problem Size\Cores	**4x2**	**6x2**	**8x2**	**16x2**	**8x3**	**12x3**	**16x3**	**32x3**
100 Miilion	6	14	23	61	21	35	29	291
250 Million	5	24	26	30	20	41	44	196
500 Million	9	24	20	86	20	37	51	470
750 Million	6	19	21	61	27	28	41	147
1 Billion	4	27	25	34	28	27	75	194

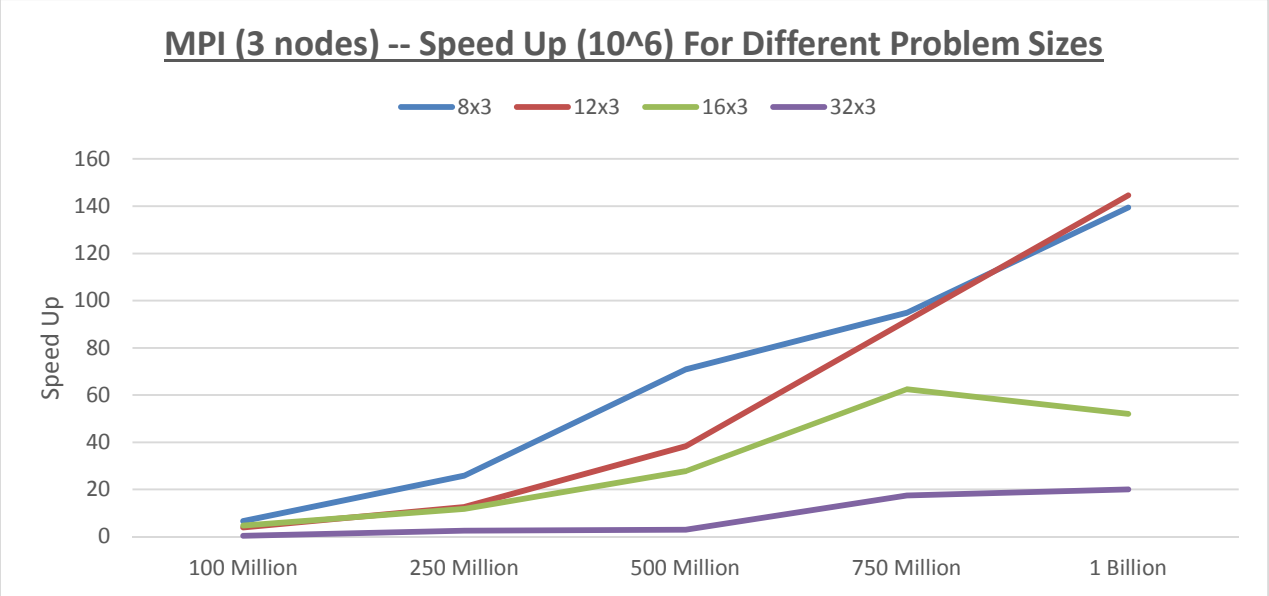




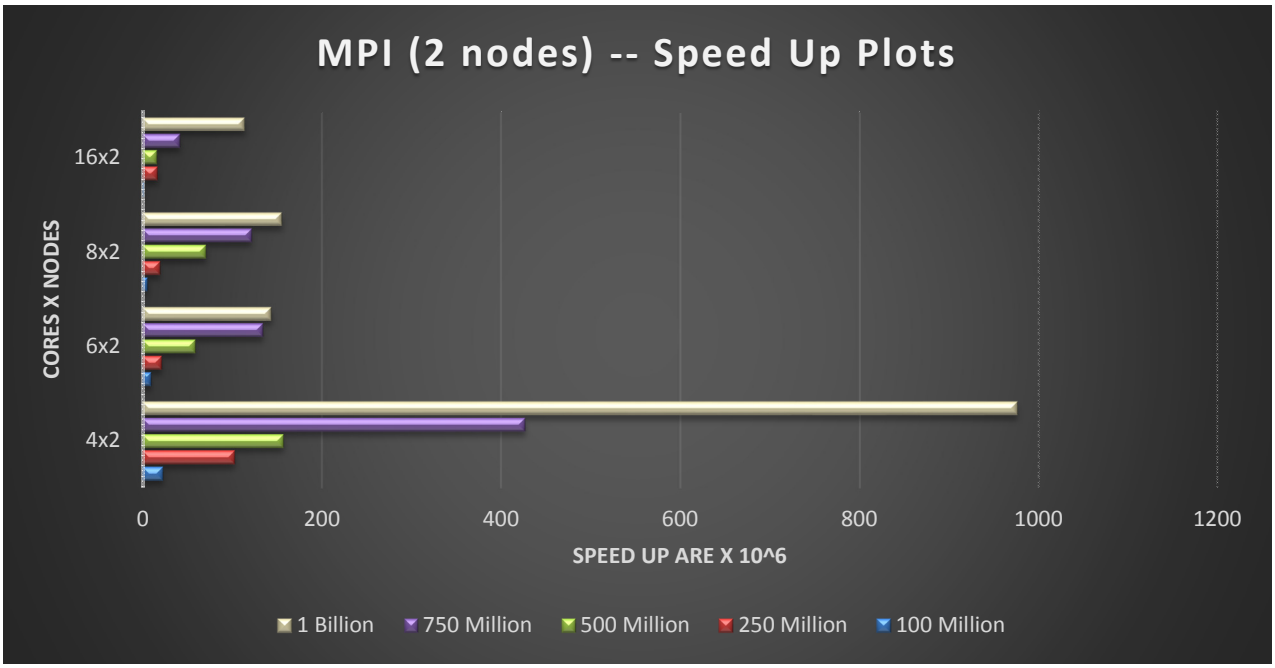
Please note the difference in scale for the two graphs above. MPI on 2 nodes with lesser number of processes performs better than MPI with 3 nodes with more processes. And also do note that the best times for MPI is given by 4 processes each on 2 nodes.

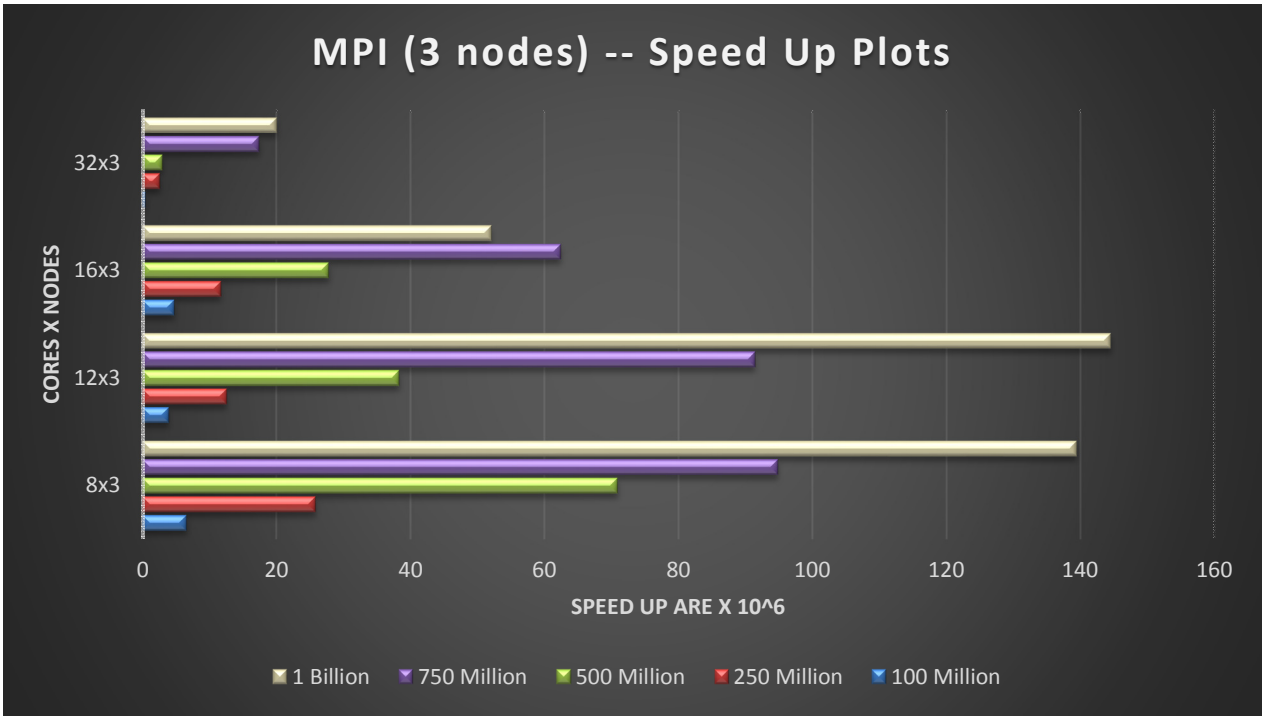
An important observation to make is that MPI times are in micro seconds and are way better than the times posted by OpenMP.





Note: Speed ups are of the order of 10^6



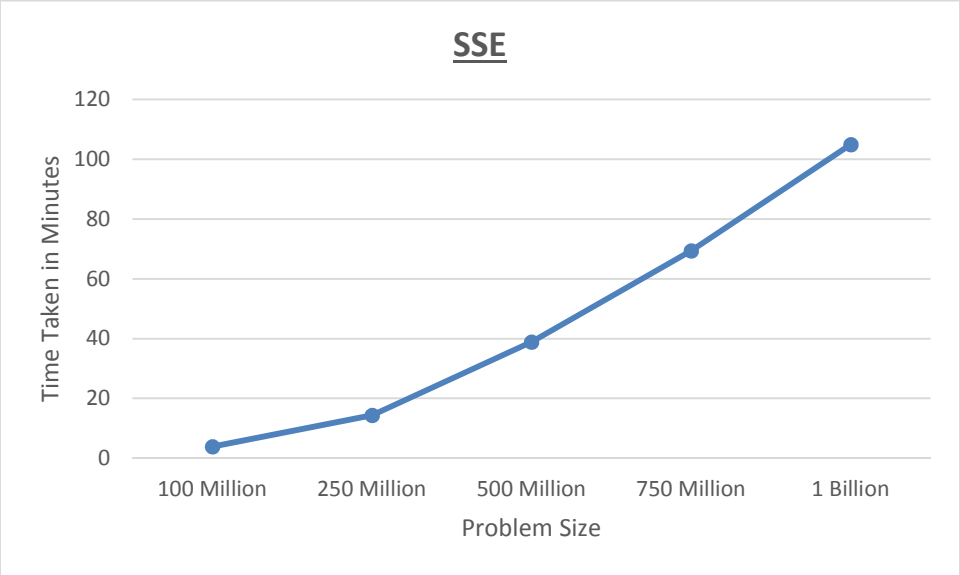


We do see massive speed ups, primarily due to turning off of print statements. Maximum speed ups are seen for Large problem size and this is almost uniform all processors x node configurations. The take away is that scaling out helps in gaining phenomenal speed ups especially when the problem size is large. Also to note here is that the increase in processors and scaling out to more nodes could be detrimental as speed ups in smaller number of nodes is much more than the combination of larger nodes and processors

SSE

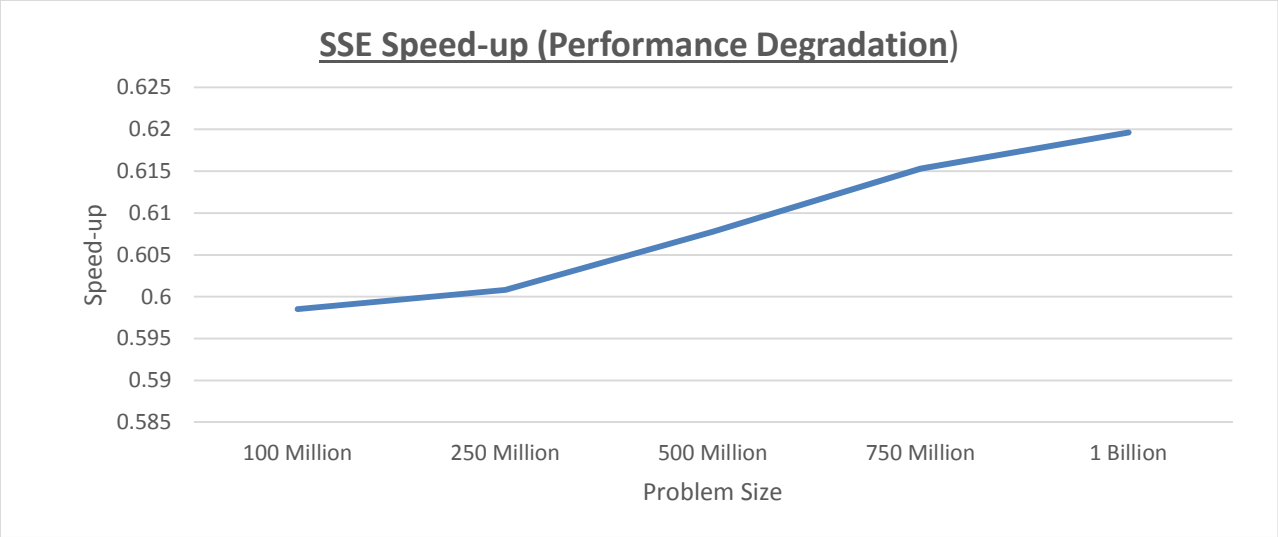
Time taken when using SSE for computing the prime numbers.

SSE		
Problem Size	**Time in Seconds**	**Time in Minutes**
100 Million	861.46	14.35766667
250 Million	3404.06	56.73433333
500 Million	9626.72	160.4453333
750 Million	17686.44	294.774



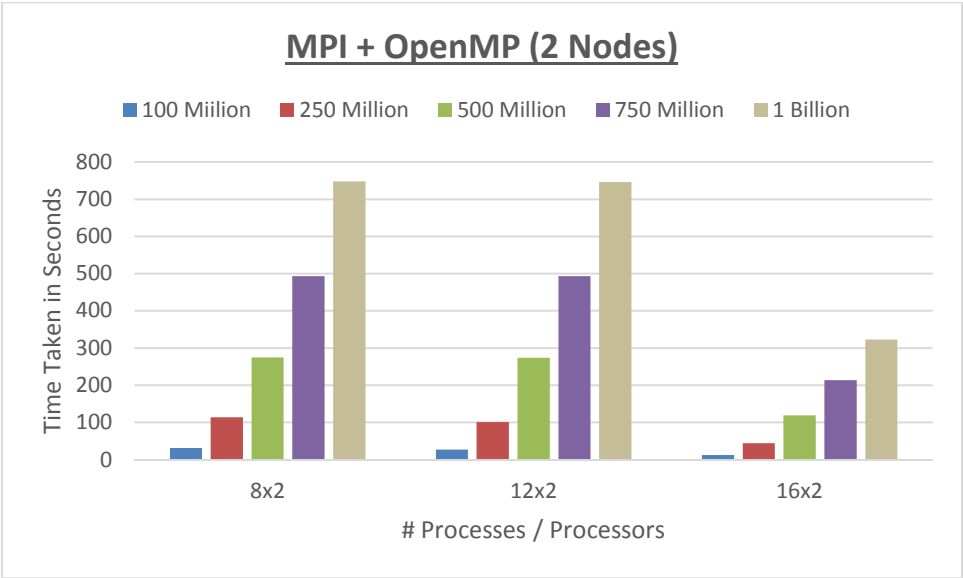
SSE	
Problem Size	**Speed Degradation**
100 Million	0.598538422
250 Million	0.600836173
500 Million	0.607788946
750 Million	0.615283243
1 Billion	0.619611008

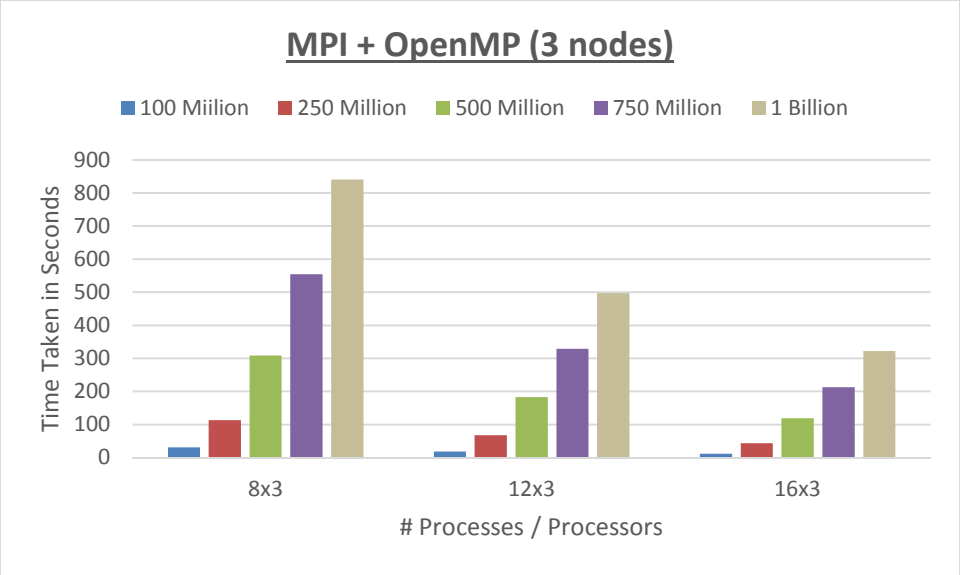
With SSE usage I was expecting 2x speed up; however we see that there is a degradation in the speed. This primarily due to the usage of double division, which is a slow operation. And the subsequent check that the code does to look for divisibility is a major factor in the degradation in speed. Intel Intrinsic are ideal for vectorization, multiplication, add, and subtract not for division.



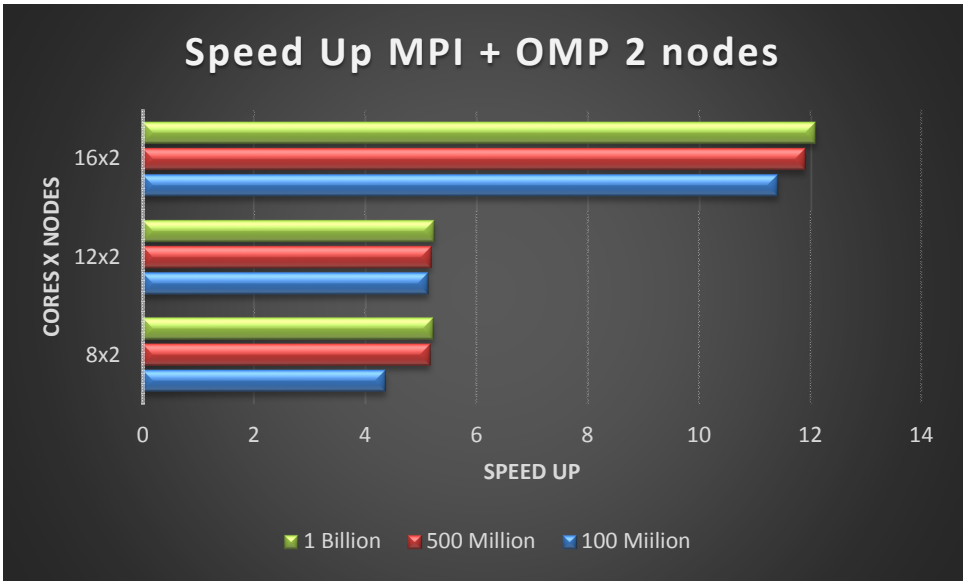
MPI + OpenMP

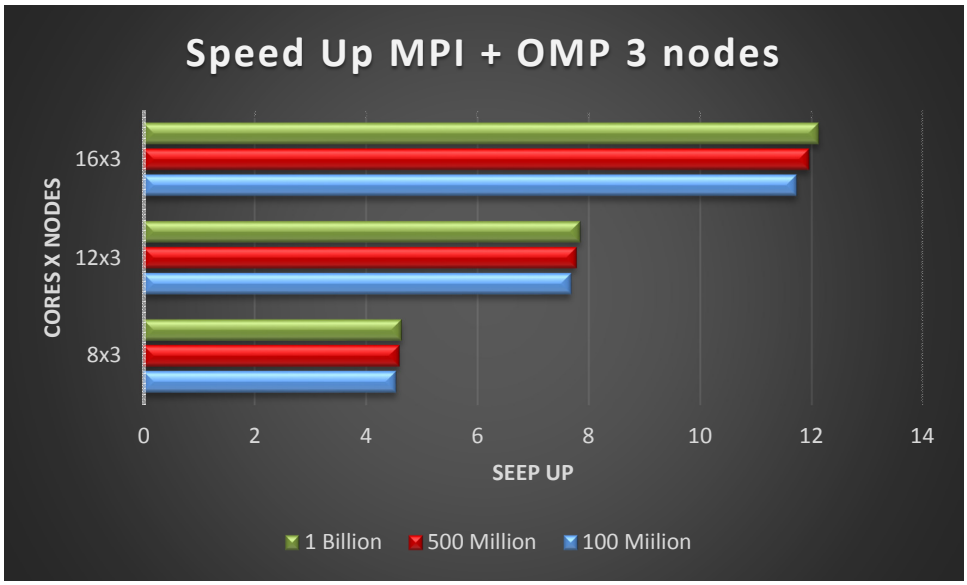
MPI + OMP Time in Seconds						
N\P	**8x2**	**12x2**	**16x2**	**8x3**	**12x3**	**16x3**
100 Miilion	31.69432	27.00126	12.12373	30.5018	18.02853	11.78153
250 Million	114.0028	100.8713	44.45979	113.5586	67.25321	43.8797
500 Million	274.7567	274.0517	119.2453	308.6778	182.7557	118.6325
750 Million	493.6098	492.8865	214.0812	554.6326	328.5962	212.7397
1 Billion	747.9086	746.186	323.0415	840.7097	497.3944	321.765





SPEED UP						
Problem Size\Cores	**8x2**	**12x2**	**16x2**	**8x3**	**12x3**	**16x3**
100 Miilion	4.361299	5.119332	11.40147	4.531811	7.667202	11.73264
250 Million	4.531599	5.121526	11.61983	4.549327	7.681643	11.77344
500 Million	5.162168	5.175449	11.8943	4.594889	7.760853	11.95575
750 Million	5.187855	5.195469	11.9617	4.617068	7.793081	12.03714
1 Billion	5.218422	5.230468	12.08174	4.64239	7.846696	12.12967





Discussion of Results

The algorithm is fairly simple and is a 'prime' candidate for parallelization. Since the whole program comprises of an iterative process it becomes an embarrassingly parallel task; however since each candidate number has to be checked till the said number's square root therefore when given a upper limit for search of primes the programmer has to be careful to spread the load for search amongst all parallel threads.

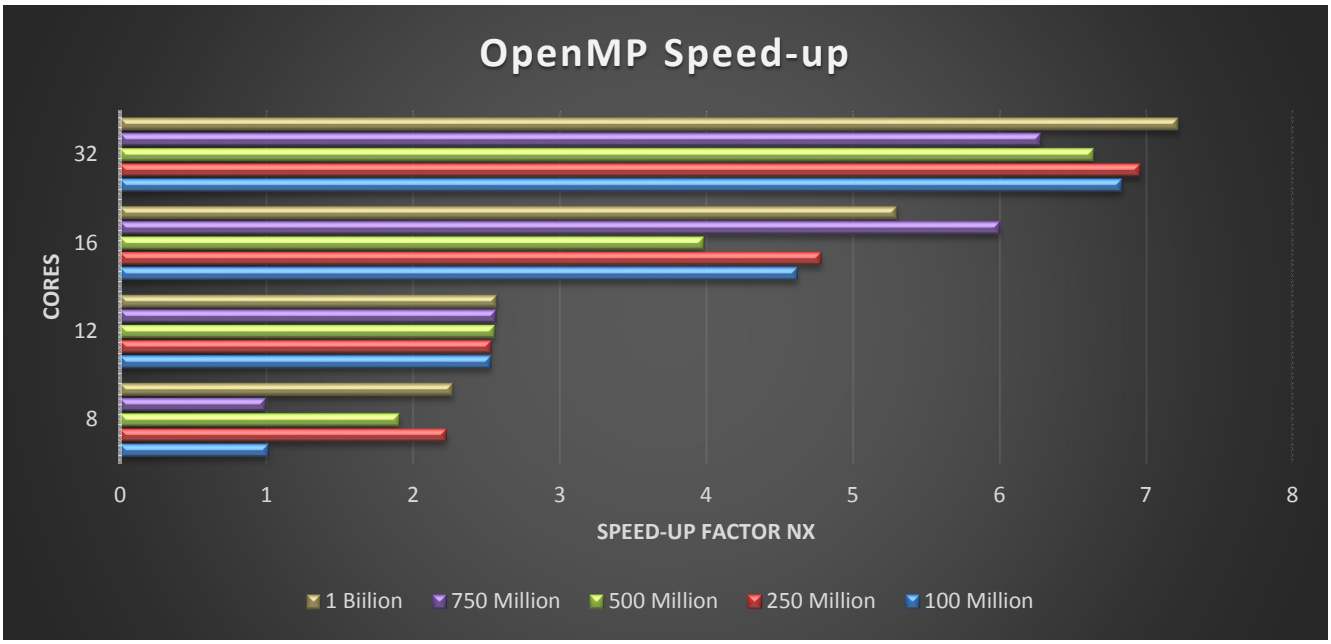
In this exercise Shared memory, Distributed memory architectures, a combination of both along with Intel intrinsic have been used to try and achieve speed ups. The speed ups in the execution times must be linear and increase with increase in the number of processors thrown at the problem.

Discussing each architecture:

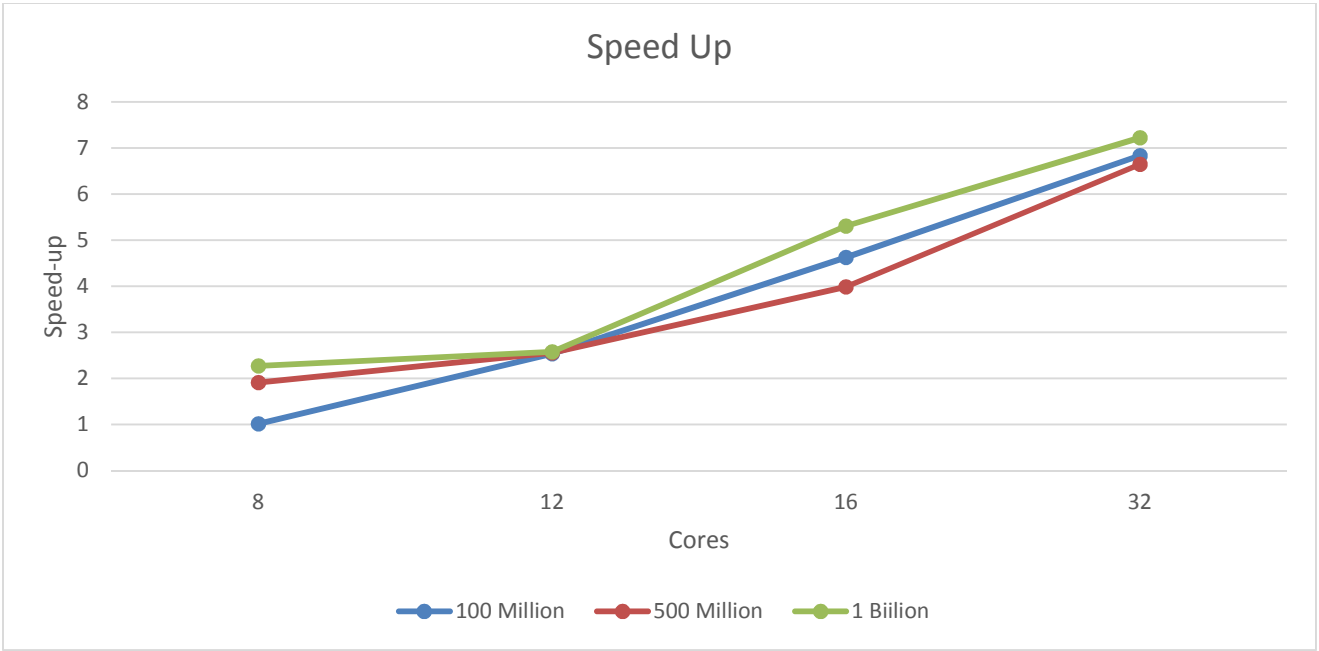
OpenMP (Shared memory architecture):

We do achieve considerable speed up while using OpenMP

- Speed up is not uniform for all problem sizes irrespective of the number of cores – Speed up is choppy, though we have good performance gain.
  - Take the case of problem size 750 Million and 16 processors we achieve better speed up for this problem size than what we achieve for 1 Billion with the same processor count
  - In the case of 32 processors and 1 Billion we have higher speed-up than we have for 750 Million
  - In the case of 8 processors we see that speed-up are seen only for problem sizes of 250 Million, 500 Million and 1 Billion



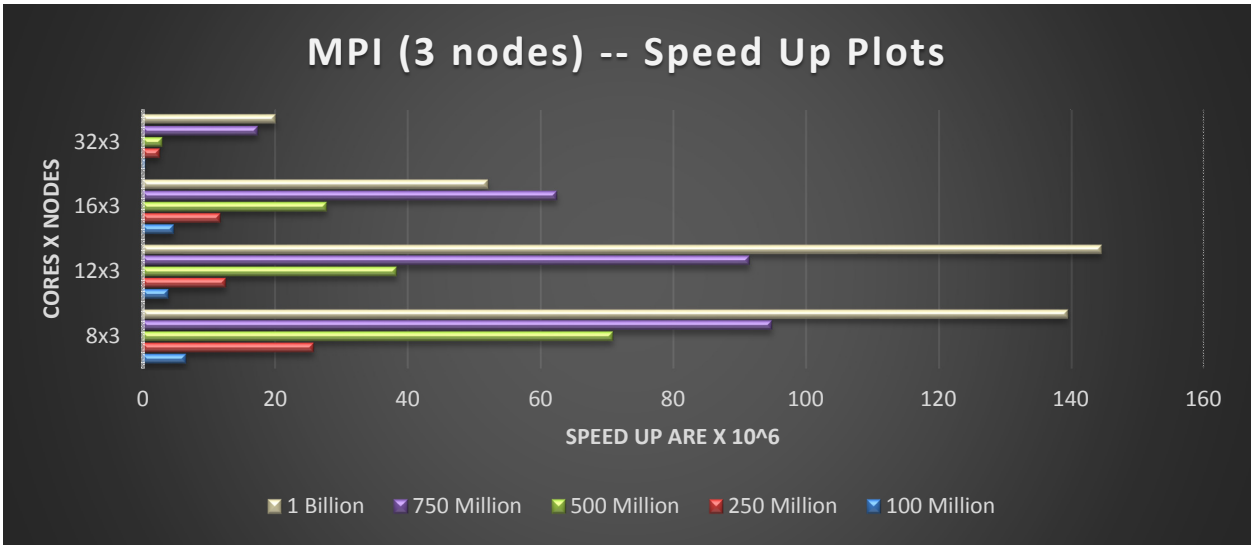
- One more important point to be noted is that using **"OMP parallel for"** shall split the loop work load in chunks amongst the processors, we have avoided that by distributing the load amongst all processors evenly
- Important Observation: Using -03 Optimization shall result in inexplicable speed-ups as can be seen from the time results for Optimized code. Do also notice that performance of optimized code on 1 processor beats the performance of the same code on multiple processors. This deviant result is taken care of when optimization is switched off.

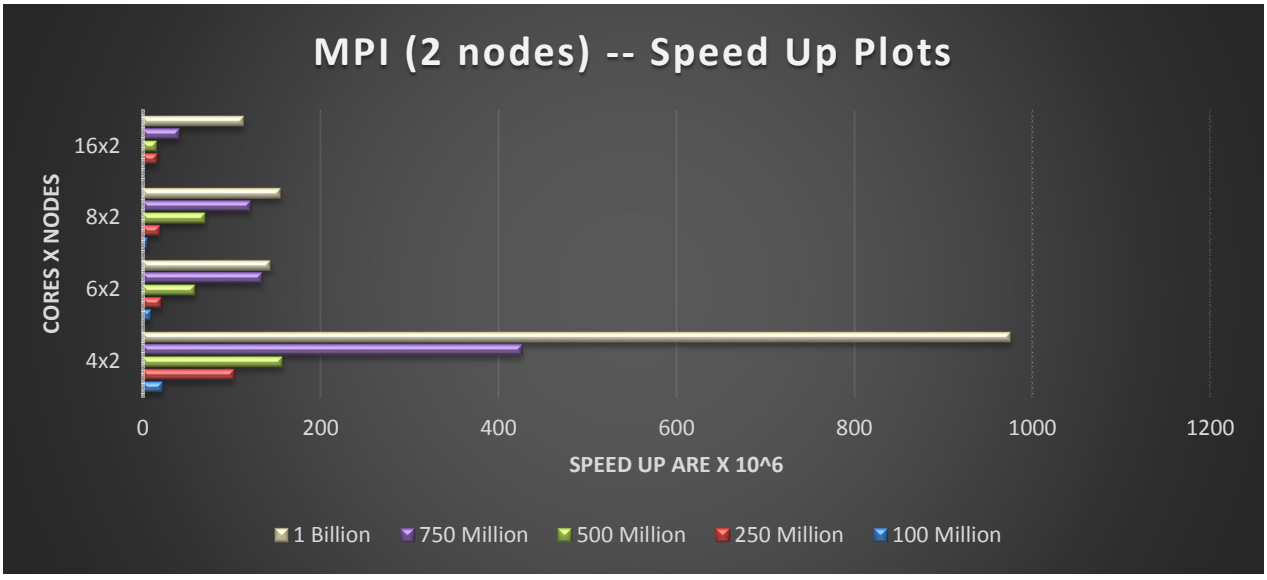


For this problem using shared memory architecture we have not been able to achieve linear speed ups.

MPI (Distributed Memory)

MPI delivers in term of speed ups. While profiling, having turned off the print statements, I observed massive speed ups w.r.t to sequential code, which is hard to believe. Since this simple algorithm does not require any synchronization and since we are not making use of the parallel file system for printing the prime numbers, we end up seeing super linear speed ups.

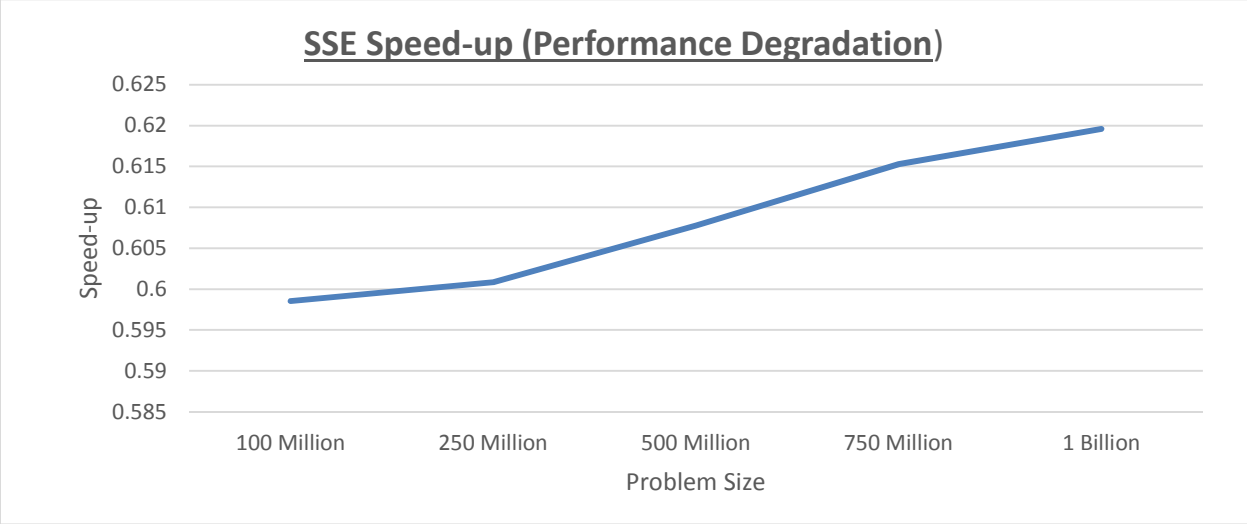




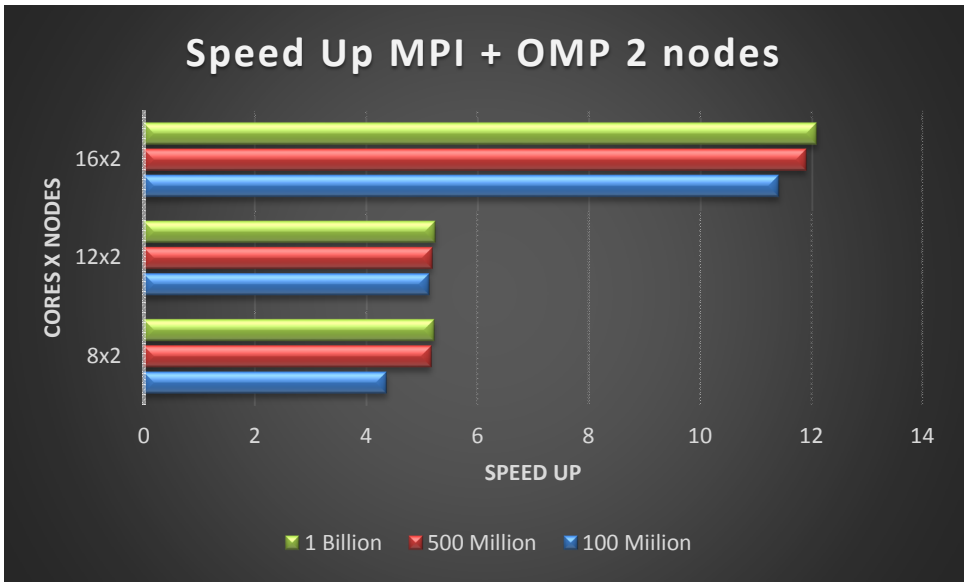
Maximum speed-ups are seen for Large problem size and this is almost uniform for all **"processors x node"** configurations. The take away is that scaling out helps in gaining phenomenal speed ups especially when the problem size is large. Also to note here is that the increase in processors and scaling out to more nodes could be detrimental as speed-ups in smaller number of nodes is much more than the combination of larger nodes and processors

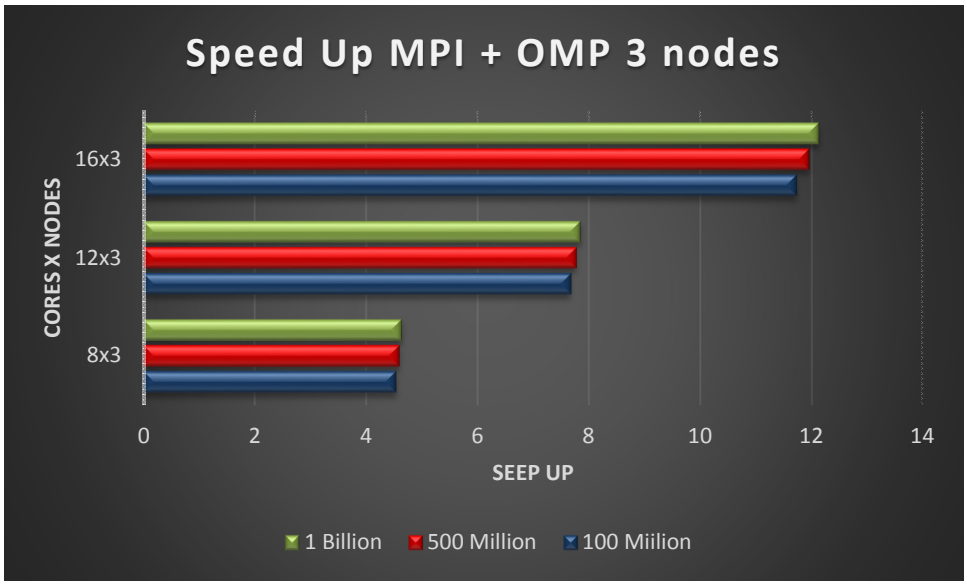
Intel Intrinsic

I used _m128d for speeding up the algorithm. However because of the operations that were required to be done, like division and which at the intrinsic level is slow and later the check for divisibility did not give any performance speed up but rather a degradation in the speeds were observed. Though the degradation in speed is lesser for larger problem sizes, one does not see any merit in using intrinsic for speed up when the problem at hand requires complex intrinsic operations like division and check for divisibility. Intel intrinsic are better for vectorization, addition, multiplication, subtraction.



MPI + OpenMP





From the above speed-up graphs one can see that a combination does provide better speed-up than stand-alone OpenMP code. Some important observations:

- The speed-up is uniform for differing problem sizes for each of the configurations that were tested
- We see better speed-ups when using larger configurations i.e 48 (16x3) and 32 (16x2). The interesting thing to note is that the speed-ups are the same for both 48 processors and 32 processors
    - Hence as we scale out and use more nodes we need more number of processor to achieve the same speed up one could achieve with lesser number of nodes and lesser processors
- Also to be noted is that 24 (8x3) processors evenly distribute across all 3 nodes gives lesser performance gain than 16 (8x2) processors which further strengthens the initial observation
- One more significant observation is that 12x3 (36) processors give better speed than 12x2 (24) processors as is expected
- Thus scaling out helps in speed-up but one needs to find the magic combination of processors and nodes to achieve this and this is dependent on the problem and the problem size both

Relation to theoretical evaluation

The theoretical evaluation of the algorithm indicates that linear speed up must be possible. 100% of the problem is parallelizable and Amdhal's law states that we should get as much speed up as the number of processors thrown at the problem. As we have seen from the results this is not possible in all scenarios and for all architectures. We do get super linear speed up using MPI but that is also due to turning off print statements.