STUDY OF SCALABILITY – GPUs

Contents

Problem Statements

Using GPUs to speed up the programs that have been built so far. The prime number generator and the Bucket sort application. We have seen that both the programs are highly parallelizable.

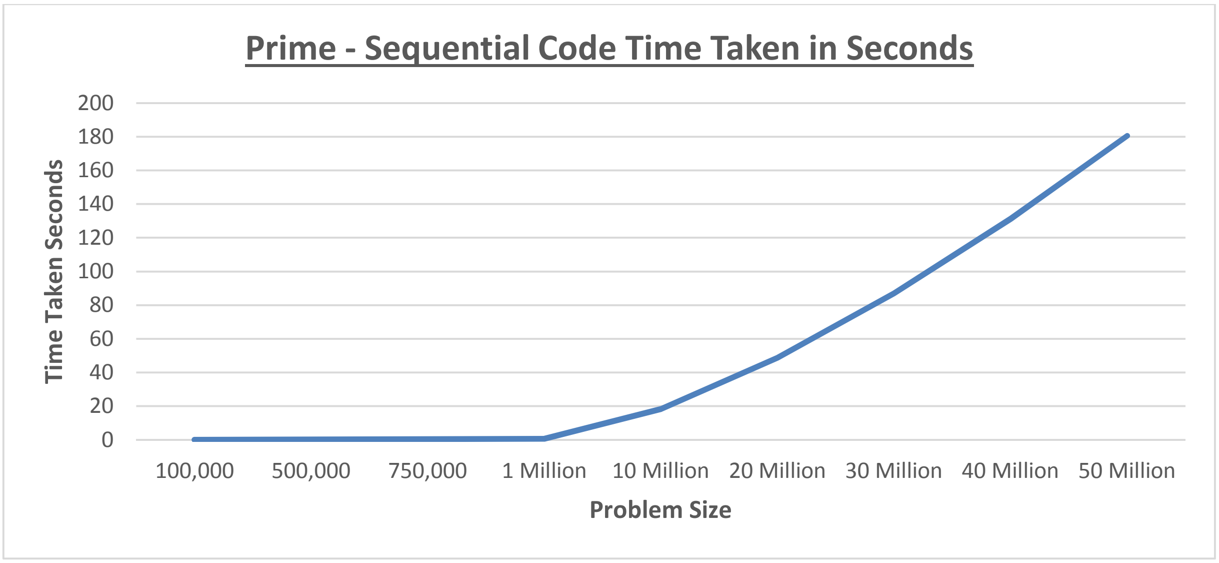
Prime number generation: The task of checking if a number is prime or not must be done by trying to divide the candidate number by all odd numbers from 3 to the sqrt(candidate) number. This task is independent and can be done in parallel.

Bucket Sort: In bucket sort the idea is to sort evenly loaded buckets and once buckets are made they can be sorted independently. This task of sorting a bucket can be done in parallel. Though the creation of the bucket is a major problem.

Sequential Code

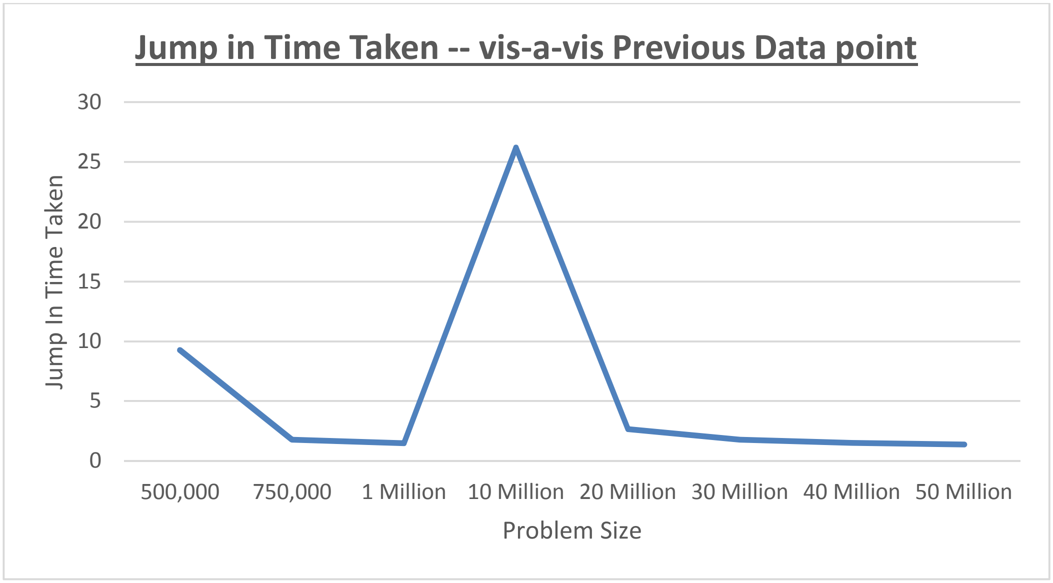
Prime number Generation:

SEQUENTIAL CODE - PRIME		
🟥	**Seconds**	**Minutes**
100,000	0.029256	0.00049
500,000	0.270858	0.00451
750,000	0.478673	0.00798
1 Million	0.700593	0.01168
10 Million	18.359368	0.30599
20 Million	48.82081	0.81368
30 Million	86.950535	1.44918
40 Million	131.297201	2.18829
50 Million	180.555513	3.00926



These numbers show how the time taken shoots up dramatically. The following table show how the time taken behaves with respect to the previous data point. We can see in the graph that follows the table that the jumps correspond to sudden increase in problem size.

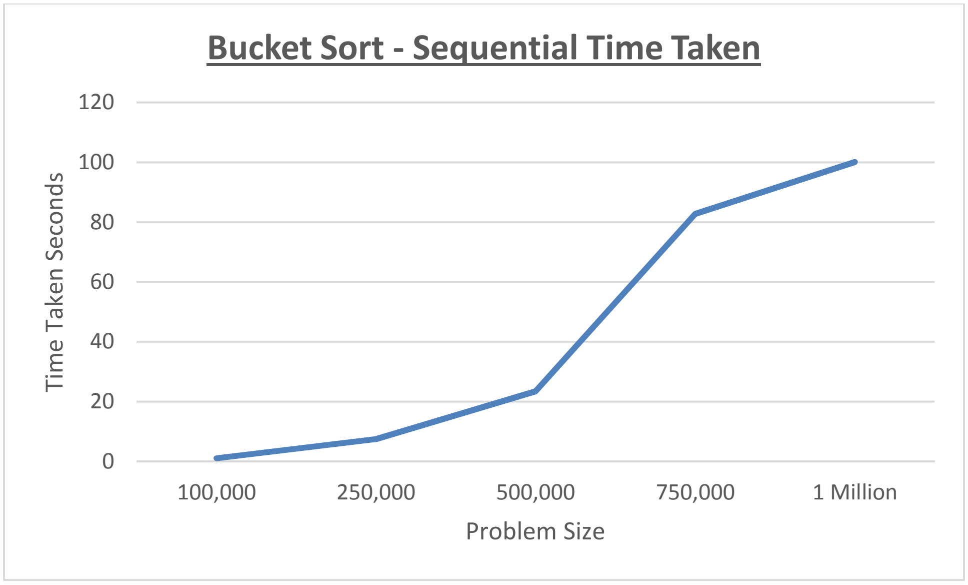
ProbSize	Delta In time
100,000	
500,000	9.258203445
750,000	1.767247045
1 Million	1.463615036
10 Million	26.2054688
20 Million	2.659177048
30 Million	1.781013773
40 Million	1.510021773
50 Million	1.375166505



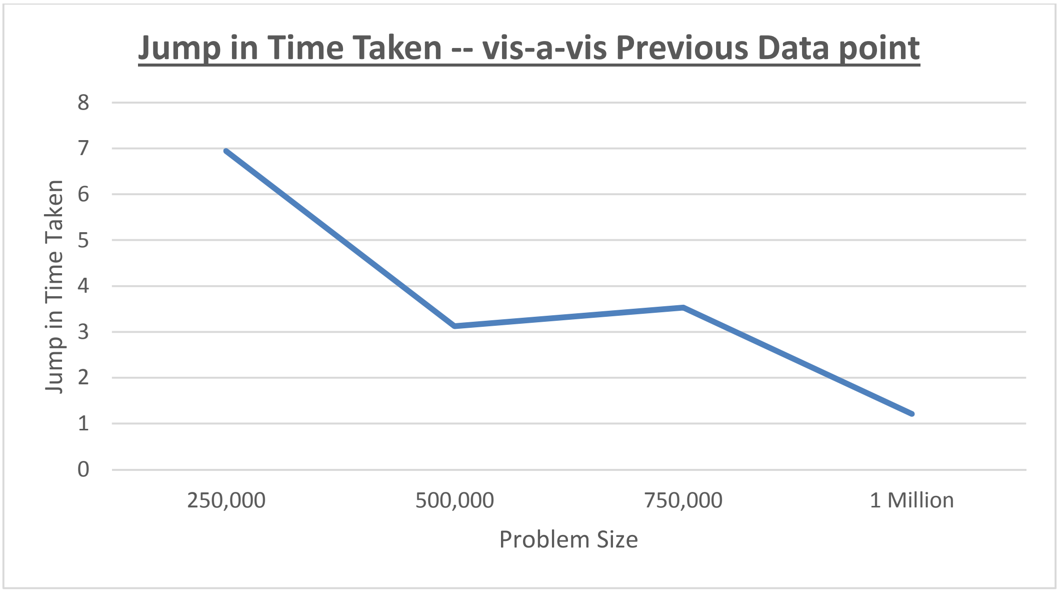
The reason for displaying the above graph and tables become apparent in the GPU section

Bucket Sort:

	Time Taken	
🟥	Seconds	Minutes
100,000	1.081097	0.018018
250,000	7.50711	0.125119
500,000	23.45483	0.390914
750,000	82.738692	1.378978
1 Million	100.092278	1.668205



Prob Size	Delta Increase in Time
100,000	
250,000	6.9439745
500,000	3.124348784
750,000	3.527575855
1 Million	1.209739671

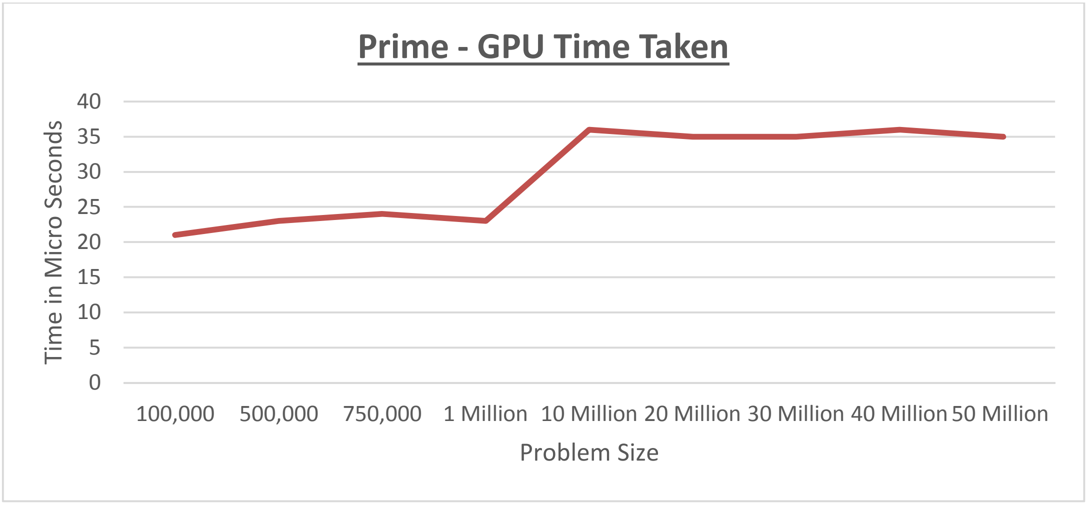


As can be noticed we see that there is an increase in time as the problem size increases.

GPU – CUDA

Prime Number Generation:

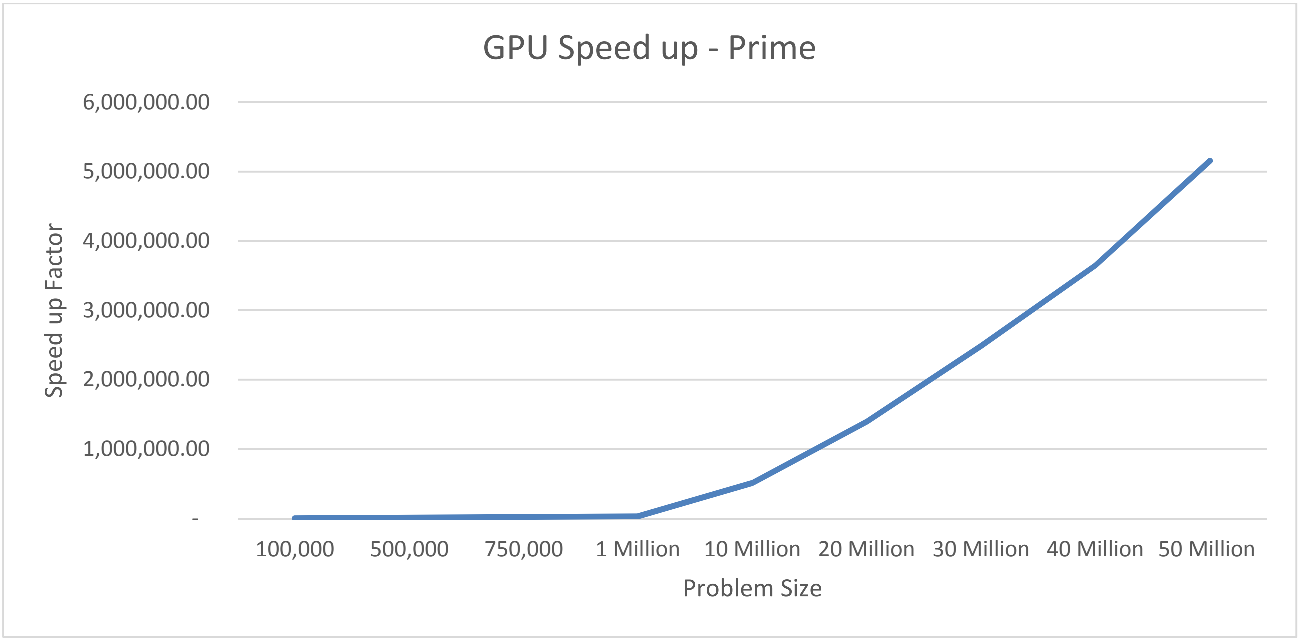
	GPU Time Taken	
🟥	Seconds	Micro Seconds
100,000	0.000021	21
500,000	0.000023	23
750,000	0.000024	24
1 Million	0.000023	23
10 Million	0.000036	36
20 Million	0.000035	35
30 Million	0.000035	35
40 Million	0.000036	36
50 Million	0.000035	35



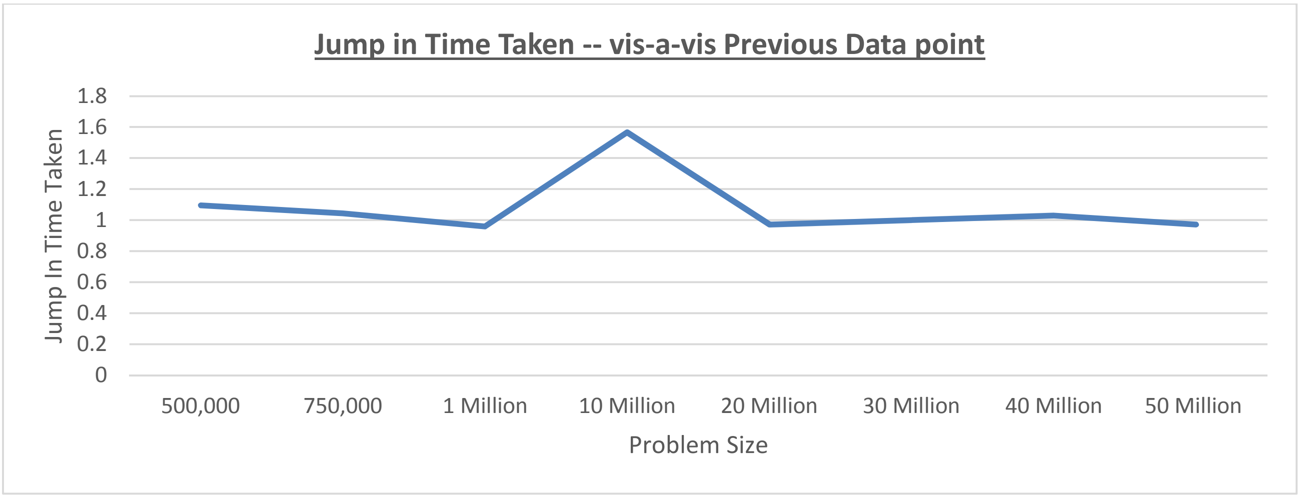
Problem Size	Speed Up
100,000	1,393.14
500,000	11,776.43
750,000	19,944.71
1 Million	30,460.57
10 Million	509,982.44
20 Million	1,394,880.29
30 Million	2,484,301.00
40 Million	3,647,144.47
50 Million	5,158,728.94

The results are quite astounding, what we see here is time taken in micro seconds as opposed to time taken in seconds by the sequential code.

Reasoning: Each block can run 1024 threads. We are working on a tesla M2050. So with CUDA prime number checker what I am making every thread in a block count. So at any given block each thread checks if a number is prime or not. This massive parallelism is the reason for this large scale speed up.



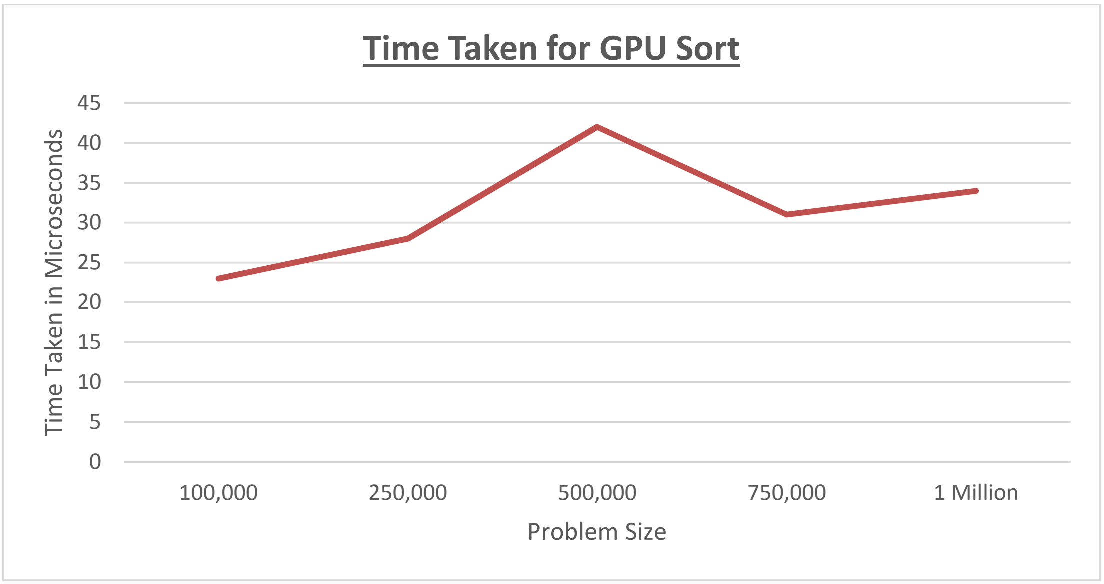
Problem Size	Increase in time
500,000	1.095238095
750,000	1.043478261
1 Million	0.958333333
10 Million	1.565217391
20 Million	0.972222222
30 Million	1
40 Million	1.028571429
50 Million	0.972222222



It is evident that the jump in time taken is almost negligible to the point that time taken becomes a constant for computation with respect to larger datasets. The reasons for this has to do with the GPU architecture and what it has to offer. One can launch roughly 65,000 odd blocks. And each block can launch 1024 threads. (50 Million / 1024) = 48,829 approximately and this helps us understand why the time taken remains a constant for all data points. If one were to increase the problem size to 100 Million then we cannot use this same program as this would mean (100 Million/1024) = 97,565 blocks which cannot be launched at one go and hence one has to change or modify the kernel launch the suite this. The above explanation gives the reason for getting near-constant time and also the reason for not testing for data points 50 Million and beyond.

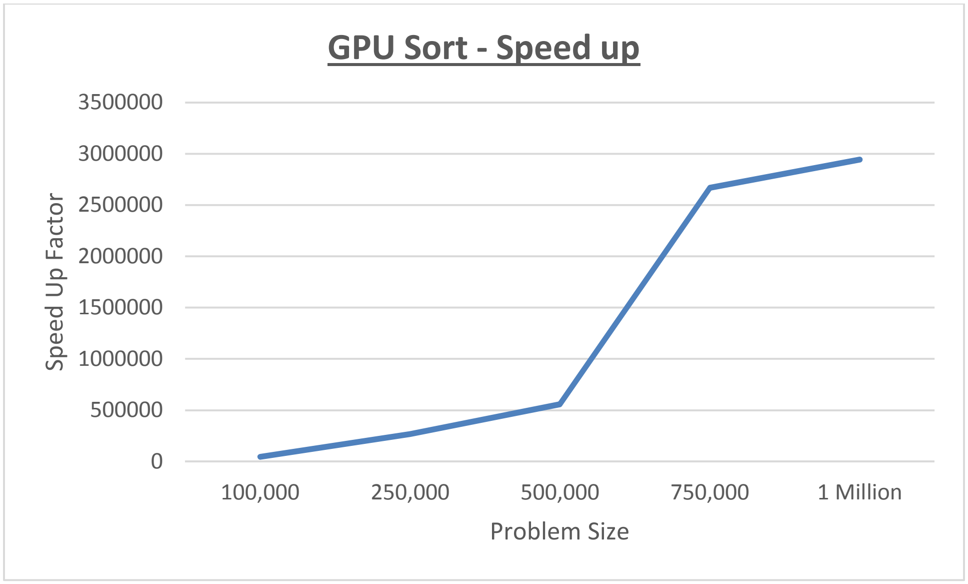
Bucket Sort:

	Time Taken	
	Seconds	**Micro Seconds**
100,000	0.000023	23
250,000	0.000028	28
500,000	0.000042	42
750,000	0.000031	31
1 Million	0.000034	34



Problem Size	Delta Increase in Time
250,000	1.217391304
500,000	1.5
750,000	0.738095238
1 Million	1.096774194

As can be noticed that we see a trend in the case above. It does take more time with increase in the problem size.



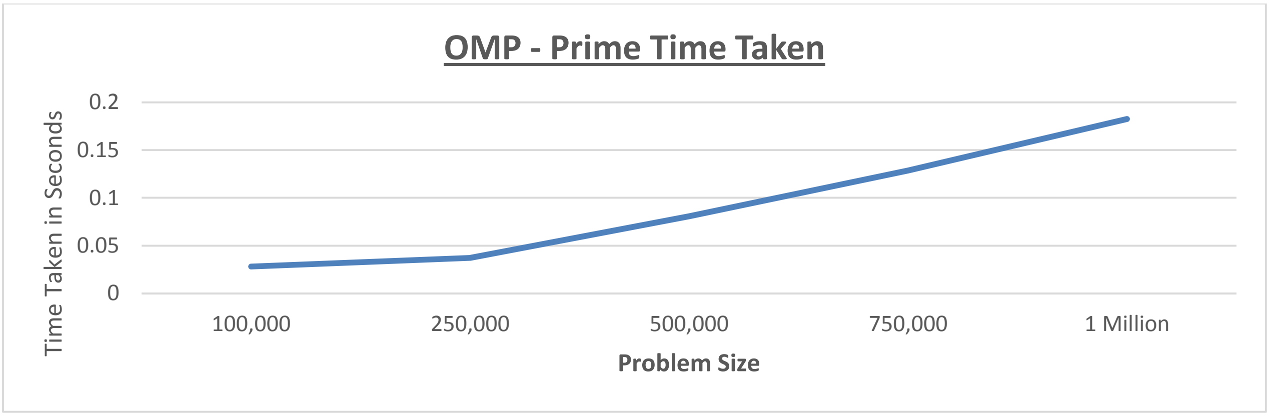
We again see massive scaling from the GPU. This is primarily due to the ability to launch many threads to do massive parallelization.

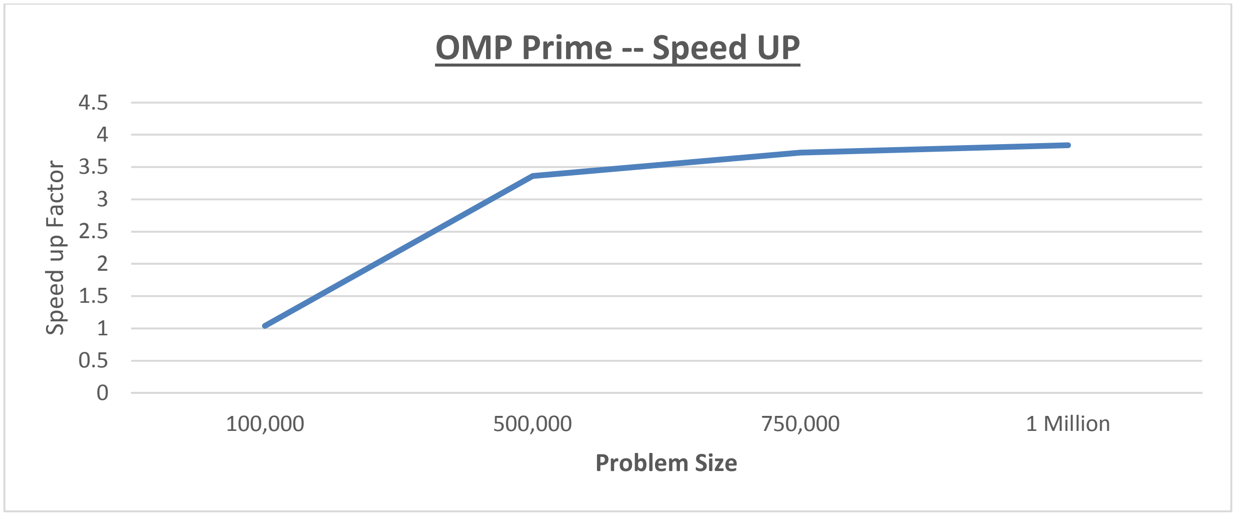
These results did seem too good to be true. I therefore had made some checks, such as for the prime numbers: Listing the largest prime number & also counting the number of primes for a given a limit. And for the sort I went ahead and checked if the output is sorted in the ascending order. The codes that have been shared have these checks and balances as part of them.

OpenMP

Prime: The following results are for OpenMP executed on a machine with 8 cores

Problem Size	Time Taken Sconds
100,000	0.028113
250,000	0.03714
500,000	0.08059
750,000	0.128532
1 Million	0.182587

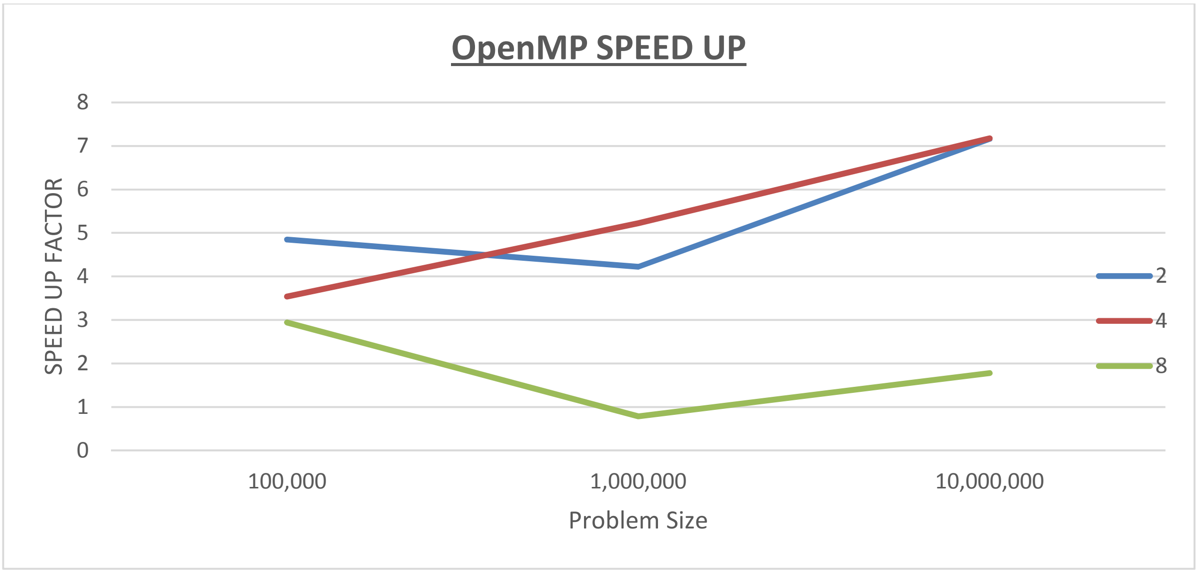




Sort: (Result from previous assignment)

$	OPENMP TIME TAKEN SECONDS		
Problem Size\ N cores	**2**	**4**	**8**
100,000	0.008354	0.011452	0.013763
1,000,000	0.120606	0.097489	0.648612
10,000,000	1.444217	1.4416	5.825037

$	SPEED-UP OpenMP		
Problem Size \ N CORES	2	4	8
100,000	4.849413	3.537548	2.943544
1,000,000	4.222501	5.223759	0.785152
10,000,000	7.162718	7.175721	1.775872

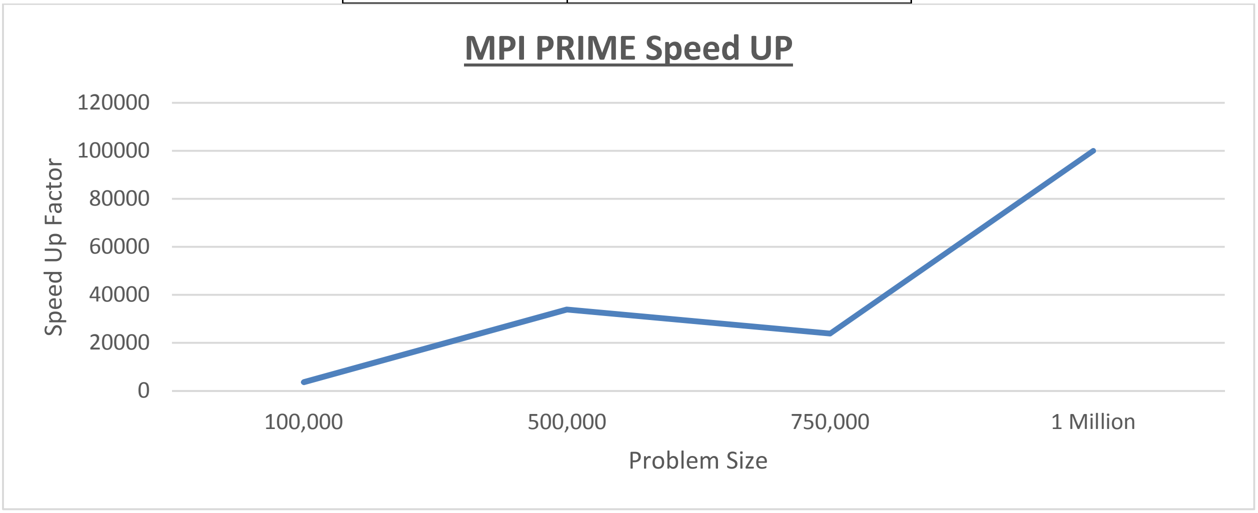


MPI

Prime:

Problem Size	Time Taken Seconds	Micro Seconds
100,000	0.000008	8
500,000	0.000008	8
750,000	0.00002	20
1 Million	0.000007	7

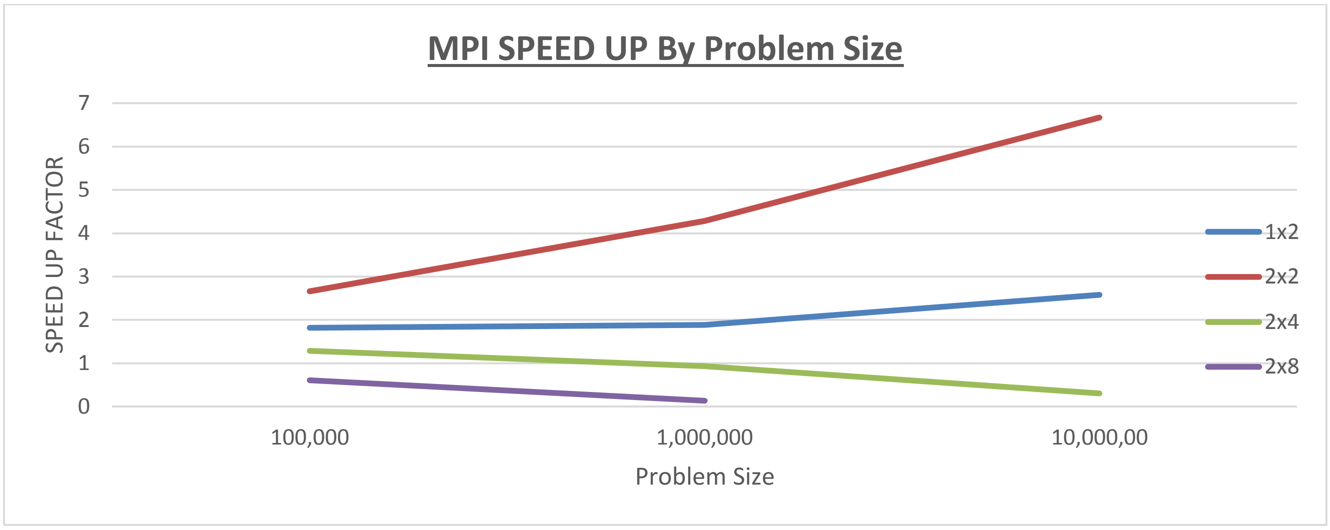
Problem Size	Speed up
100,000	3657
500,000	33857.25
750,000	23933.65
1 Million	100084.7143



Sort: (Result from previous assignment)

$	MPI SCALING OUT (Include Time for Data Transfers)			
Problem Size \ N Cores	**1x2**	**2x2**	**2x4**	**2x8**
100,000	0.022308	0.015233	0.031467	0.066356
1,000,000	0.270464	0.118943	0.547569	3.788662
10,000,00	4.006131	1.551939	33.41657	**DATA NOT AVAIL**

$	SPEED - UP - SCALE OUT			
Problem Size \ N Cores	**1x2**	**2x2**	**2x4**	**2x8**
100,000	1.816030124	2.659489267	1.287443989	0.610525047
1,000,000	1.882908631	4.281538216	0.930036215	0.134416583
10,000,00	2.582171926	6.665544844	0.309562561	**DATA NOT AVAIL**



Discussion of Results

GPUs with their massively parallel architectures make scaling very easy and also very beneficial. From the experimental results it is clear that the cost benefit analysis tilts the scales in favour of GPUs.

We can see that it is only in the prime number case that GPUs are given a stiff competition from the MPI (scaling out architecture). But however I have to account for the fact that I am not considering data transfer times between device and host either ways. So the final conclusion is that

The reason for what are witnessing:

1. GPUs are better than OpenMP by the sheer number of thread that are at their disposal
2. MPI though does compete well with GPUs but GPUs shall perform better when the data being operated on is small in size and can be held with-in the memory of the GPU then GPUs

beat out MPI dues to higher number of threads that can be spawned and the quick data transfer

a. However MPI shall have the last word when the data size that being operated on becomes too Huge.