STUDY OF SCALABILITY

BUCKET SORT

Contents

Bucket Sort

Input: An unsorted array of real numbers.

Input data Property: The numbers are generated using a random number generator which churns out numbers using the parameters of a normal distribution.

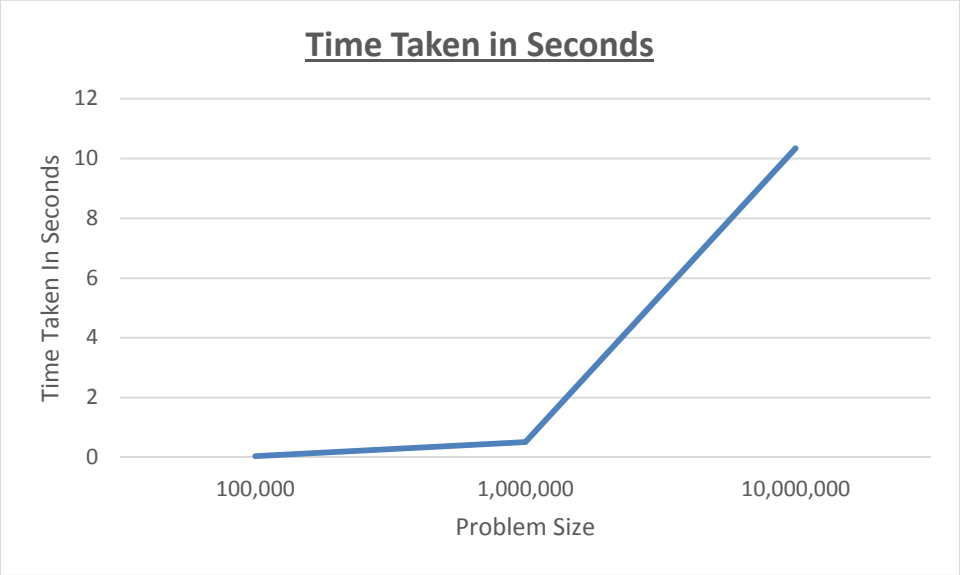
The general idea is to partition the number into buckets. Since the data is normally distributed, creating fixed with buckets shall results in unevenly distributed buckets, and also the number successive buckets have to be greater than all numbers in the previous bucket.

Maintaining order amongst the buckets: One means of achieving order among buckets is to use a hash function which shall do the job of allocating numbers to buckets. However even such a function may not help in this case as we are concerned with load balancing.

Therefore as a design choice, I had used code from the net to ascertain Kth smallest element in an unsorted array and used the statistic to populate the buckets. The time taken for this task is not accounted for in the measurements (Indeed it should not be)

Sequential

SEQUENTIAL TIME TAKEN	
Problem Size	**Time Taken in Seconds**
100,000	0.040512
1,000,000	0.509259
10,000,000	10.344519

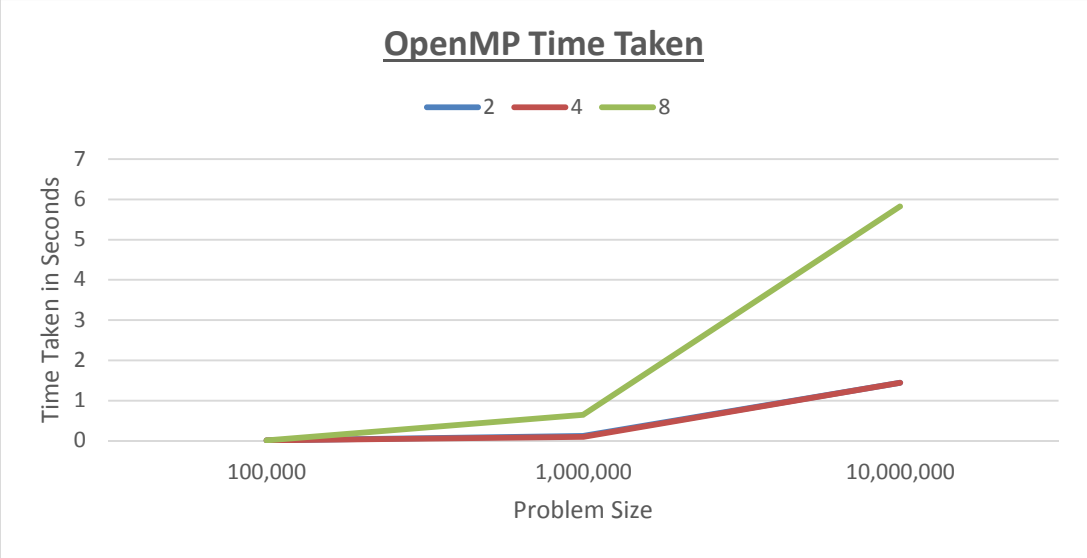


Rationale for Parallelizing

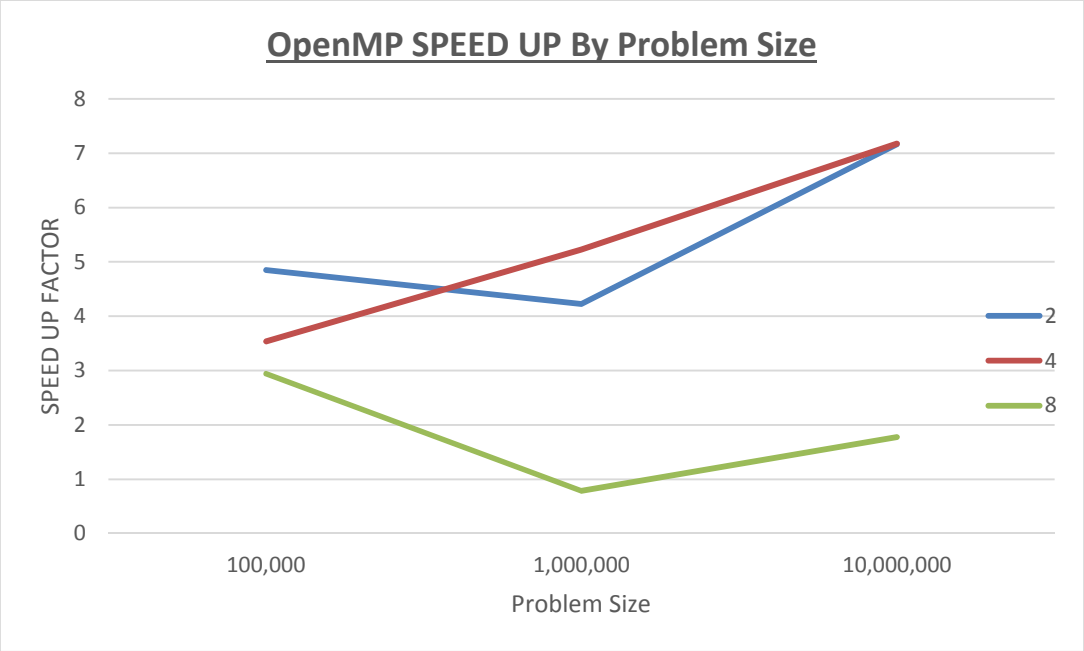
Bucket sort lends itself to parallelization. Since we have are portioning the data into ordered buckets and then sort them individually, we could hand of the sorting tasks to idle processors with-in a machine using OpenMP or have the buckets handed over to processing running on other machines via MPI.

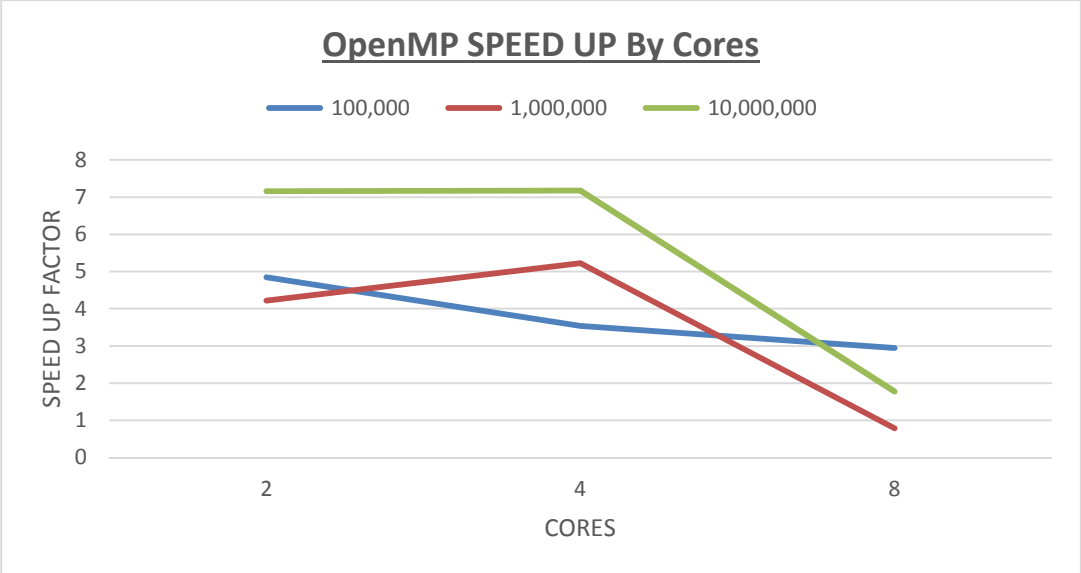
OpenMP – SCALE UP

$	OPENMP TIME TAKEN IN 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



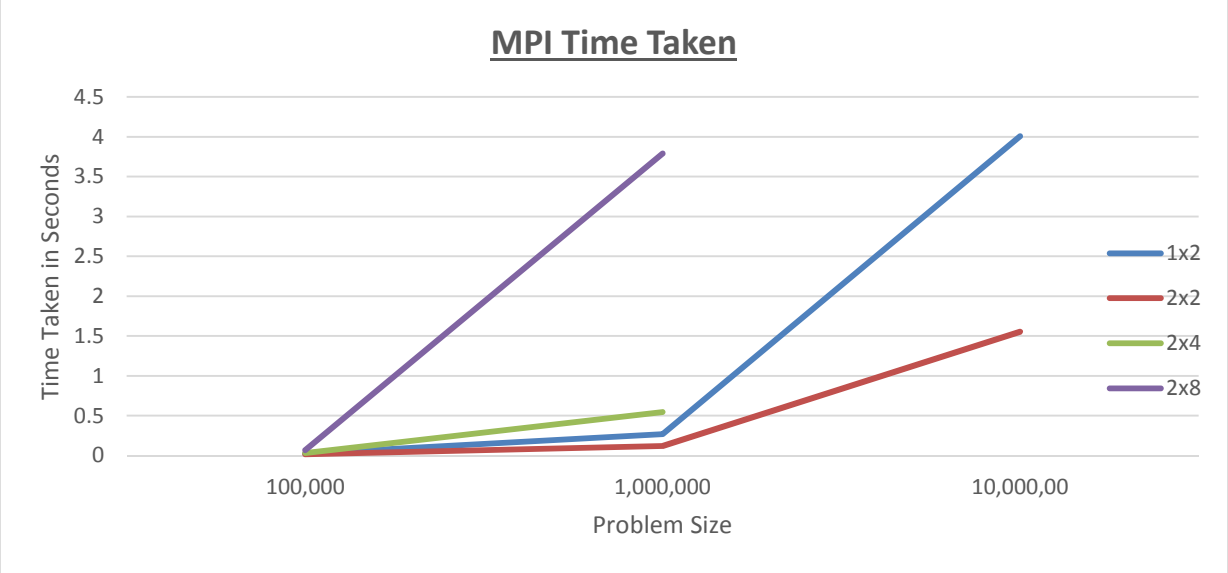
$	SPEED-UP OpenMP		
Problem Size \ N CORES	2	4	8
100,000	4.849413	3.537548	2.943544
1,000,000	4.222501	5.223759	0.785152
10,000,000	7.162718	7.175721	1.775872



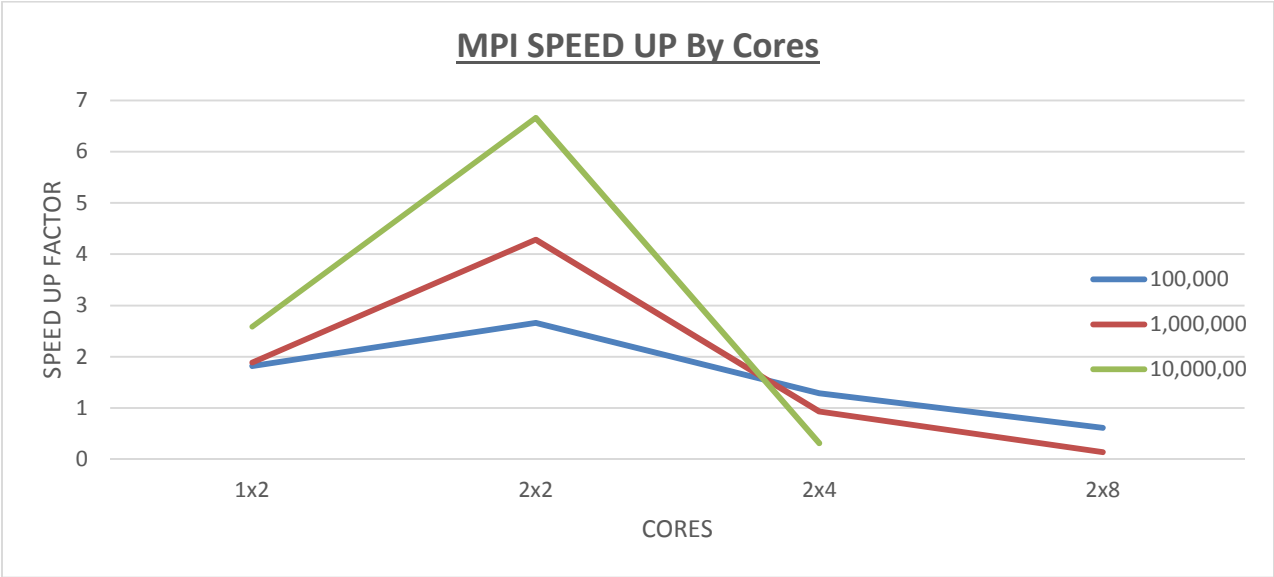


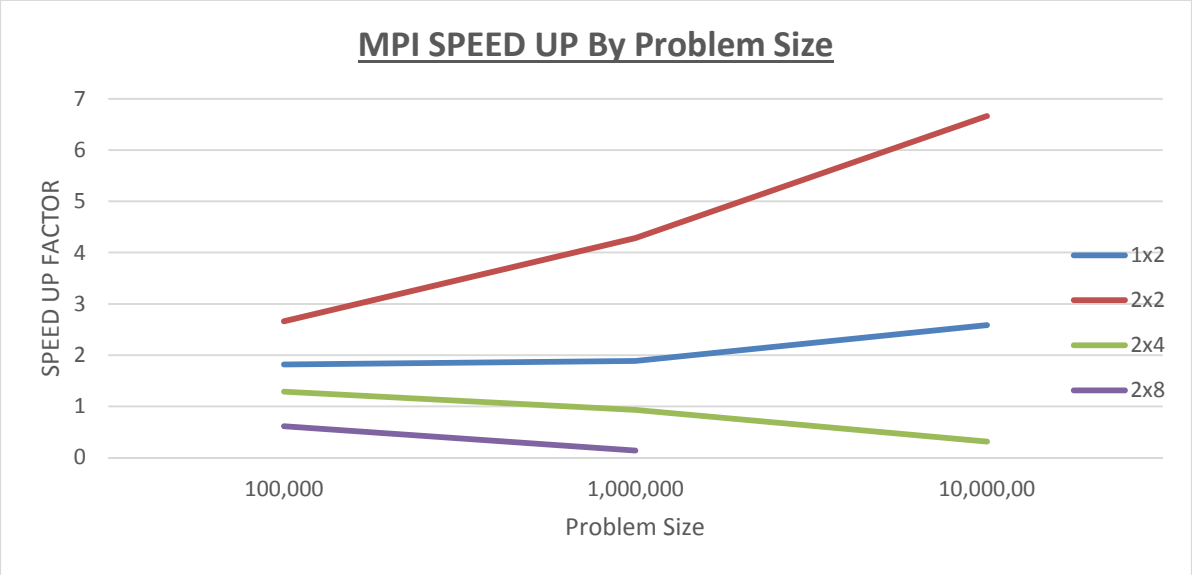
MPI – SCALE OUT

$	MPI SCALING OUT (Include Time for Data Transfers) TIME IN SECONDS			
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**



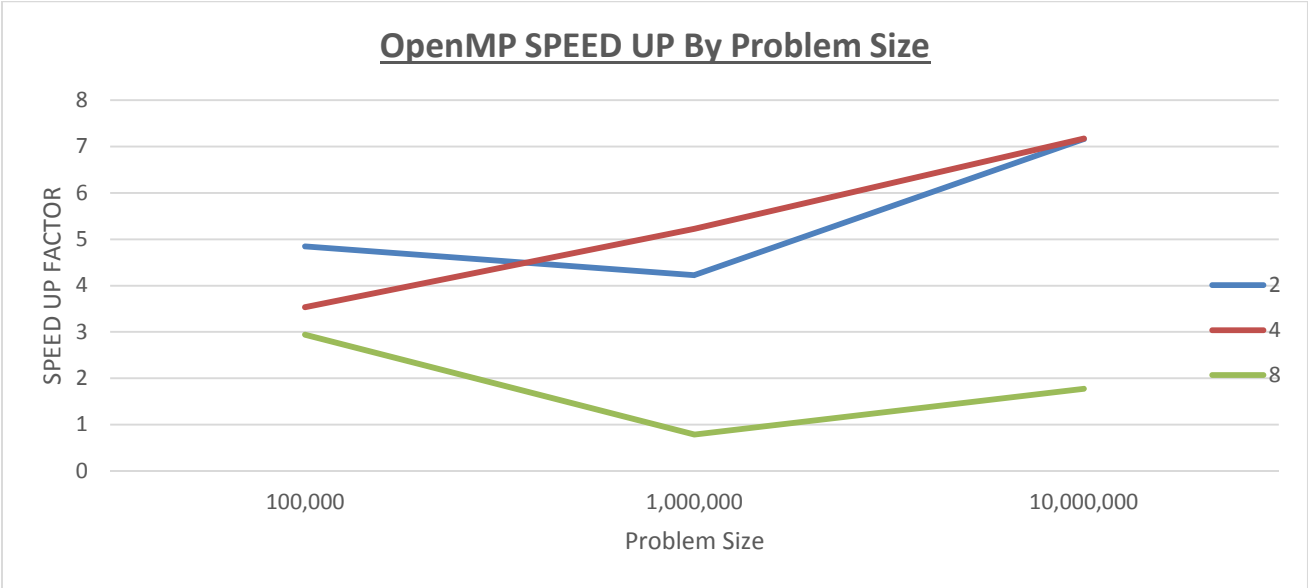


Conclusions:

The program realizes good speed ups when parallelized using OpenMP and MPI.

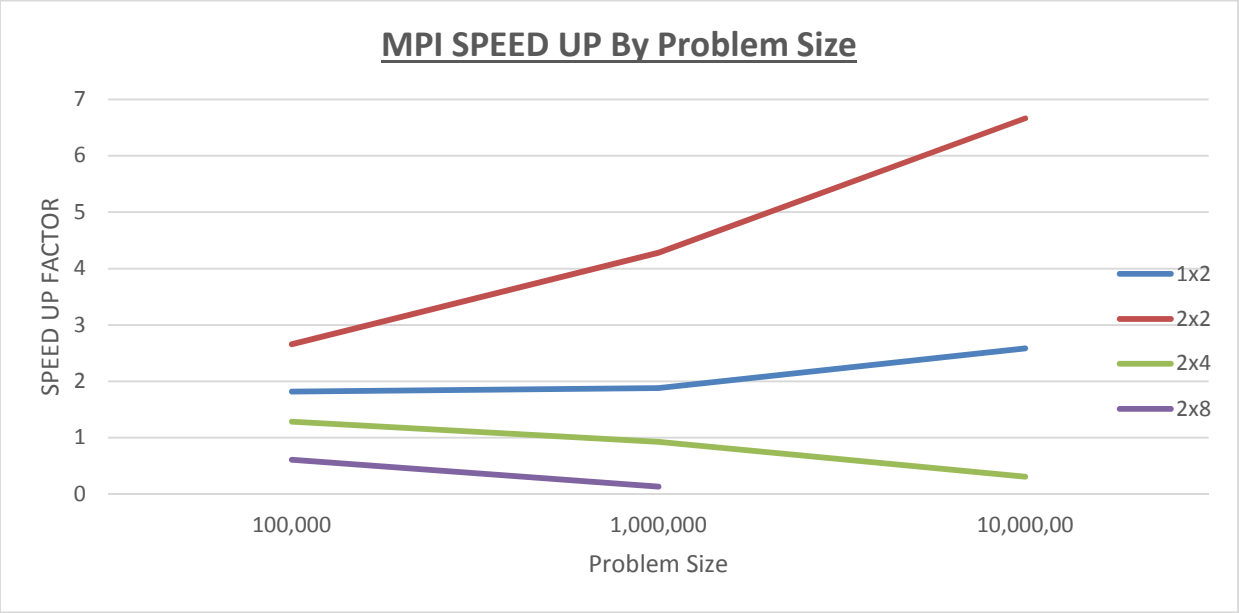
SPEED-Ups with OpenMP (SCALE UP):

1. Speed-up is not uniform – Varies for different problem sizes
    a. Cores Constant : We see higher speed up for larger problem sizes, in-fact highest speed up for the largest problem size 10,000,000; 7x speed up (2 and 4 core configuration)
    b. Problem Size constant : Increasing the number of cores does not give speed-up in-fact performance degradation is observed
2. # Cores 4 : We observe increasing speed-up with increasing problem size. In this scenario 4 cores are optimal



Speed-up MPI (Scale Out):

1. Speed-up is uniform for data-points for which it is true
    a. Keeping # Cores constant: for the configurations 1x2 and 2x2 we see that speed up of 2x and close to 7x in the case of 2x2 config.
    b. Keeping Problem Size constant: We notice that increasing cores does not necessarily give us speed-up the configuration 2x2 seems to be the inflection point since it gives better result for all problem sizes. However increasing the cores beyond 2x2 and using configs 2x4 (8) and 2x8 (16) result in performance degradation
2. 2x4 (8) & 2x8 (16): For all problem sizes give sub-par performance, if we were to use sequential times as a frame of reference
    a. Reason: With increase in the number of processors, we increase the number of buckets as well. Though the buckets are evenly created, the sheer amount of data passing from one to all and all to one causes this problem
    b. The data send and receive are blocking calls and end-up doing implicit synchronization at the master causing extra time to be taken
    c. Even though increase in the number of processors means smaller buckets and therefore less data transfer per send and receive; this acts counter intuitively as the master has to wait for that many sorted buckets to put out output



Impact of load balancing:

- Gives better performance, as opposed to un-evenly balanced tasks being done in parallel as that shall result in speed being attenuated based on the slowest process
- Distribution of work amongst processor does not give performance gain at all times. There is an inflection point when the performance starts degrading and the parallel program becomes slower the serial one

Comparison with Theoretical Evaluation:

- We see better performance for larger problem sets validating Gustafson's Law.
- Amdhal's law does not hold good for larger problem sets
- Increasing the number of processors does not give increase in performance. In the sense the relationship between processors and speed-up is not linear.
    - Instead we see inflection points beyond which performance degrades.