ASSIGNMENT 2 – REPORT
CSE603

Anand Tiwary

UNIVERSITY AT BUFFALO [Company address]

Contents

Introduction

This document presents the results of Programming Assignment 2. I have implemented Bucket Sort using Sequential, OpenMP, MPI and OpenMP + MPI. This document presents various observations and inferences obtained from these approaches.

Sequential

1. **Approach:**

In this approach, I am dividing the input numbers into buckets of equal size using the Selection-Partition Algorithm. The details of this algorithm strategy are presented below. In Sequential approach, buckets of equal size do not make much difference since the operations will be performed by one processor, so no speedup will be obtained. But, this approach would make it easier to parallelize with OpenMP, MPI and OpenMP + MPI. It will also provide common ground on which we can evaluate their relative performances. Details of this approach are as follows:

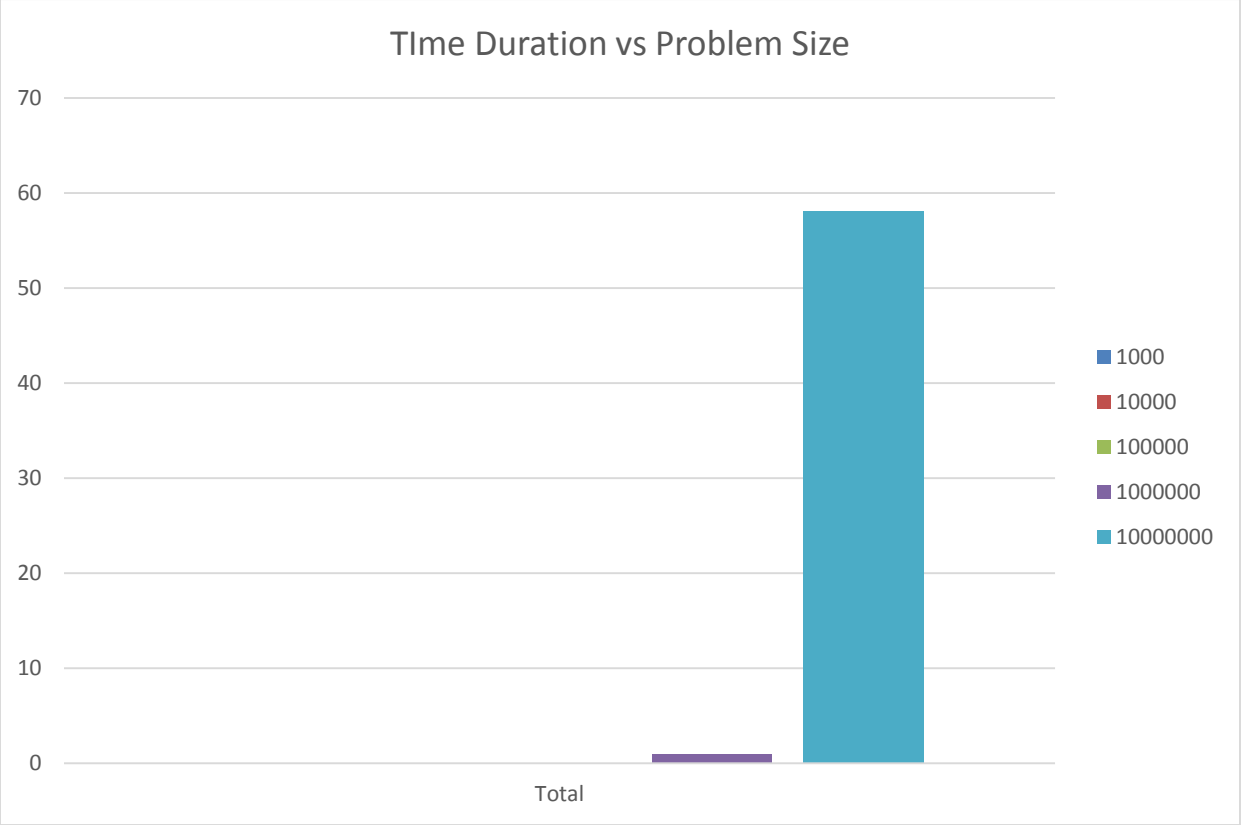
- **Partition Algorithm**: This algorithm takes the last element of the array and rearrange its position such that at new position, all elements on the left of it are smaller than this number and all number on its right are greater than or equal to.
- **Selection Algorithm**: This algorithm takes as input a '**Position'** inside the array(index) and calls partition multiple times such that after it terminates, the element at the '**position'** index is greater than all elements towards its left and smaller than all elements towards its right. The elements in the left and right sub-array are 'unsorted' but they ensures the property with the element at 'position' index stated above.

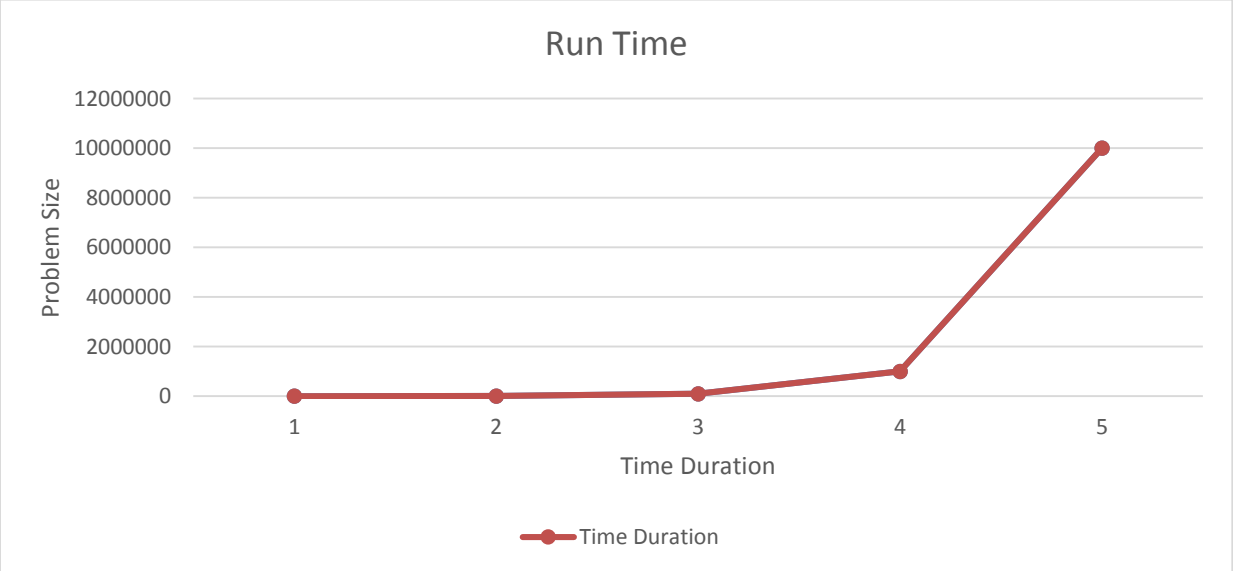
With these two algorithms, I created buckets of equal size using the below approach.

- Take the input size and divide it by number of available processors. For Sequential, I have fixed this value to be 10. The value after division will be the **Blocksize**.
- For each block, call Selection algorithm on the entire array passing the (Block-number* Blocksize) as the position. This will ensure that each block correctly contains only the numbers which are greater than numbers in previous blocks and smaller than those in the next blocks.
- After calling this selection algorithm for each block, we end up with buckets of equal size.
- These buckets are then sorted by the program one by one, and results are written back to original array in the same order. In the case of Sequential, this is done in place.
- I have used 'Quicksort' Algorithm for sorting the buckets, utilizing the Partition algorithm already created.
- I am using Randomized version of Selection-Partition to have a quick termination in Theta(n) time in average.

2. Graphs:

- **Run Time Growth:**





2. **Performance Evaluation:**

- **Increase in the size of the problem**:

  We can observe from the graphs above, that the run time increases exponentially with the increase in problem size. For 10^7, the sequential program took 52 seconds. When I ran the same for 10^8, the program took more than 15 minutes. This this approach is not scalable and efficient for large problem size.

- **Increase in the number of cores**:

  In this case, there will be no performance gain if we increase the number of cores. Since this is a sequential program, all the operations are performed by one Processor.

- **Impact of load balance on performance:**

  In this case also, since there is only one processor, we are not getting any performance improvement by load balancing i.e. by dividing data into buckets of equal size. I can say that the load balancing had result in a slightly worse performance due to the added overhead of Selection – Partition algorithm

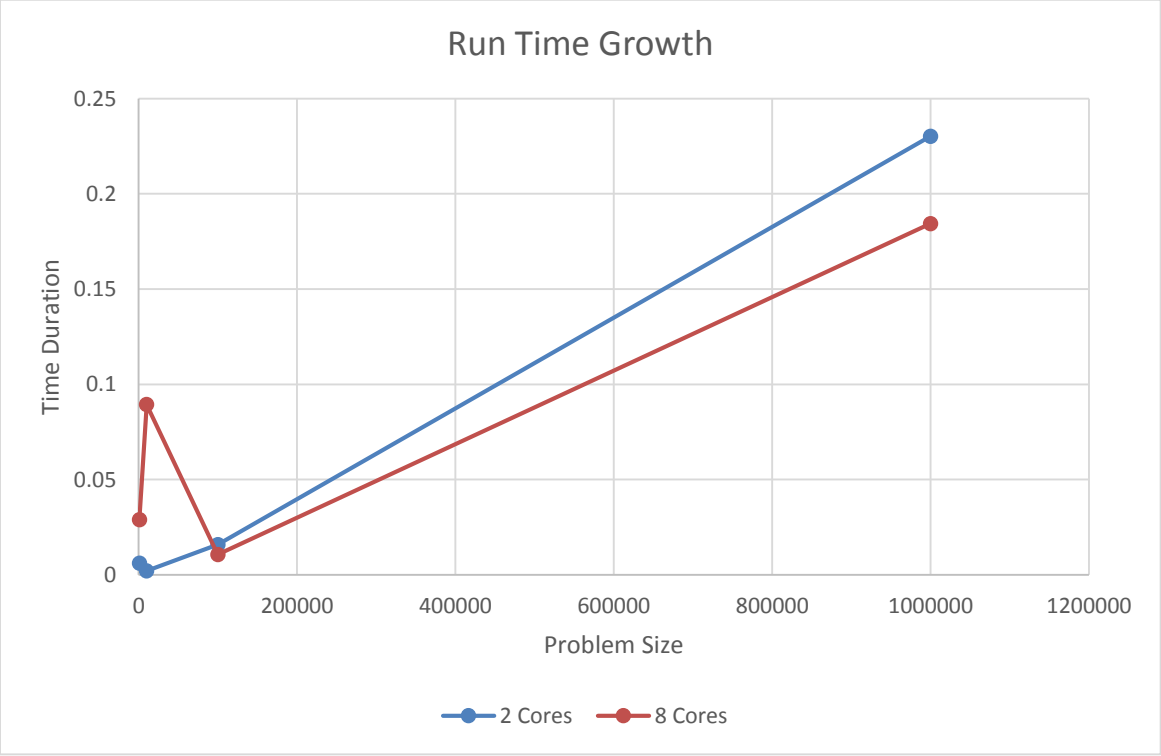
OpenMP

1. **Approach**:

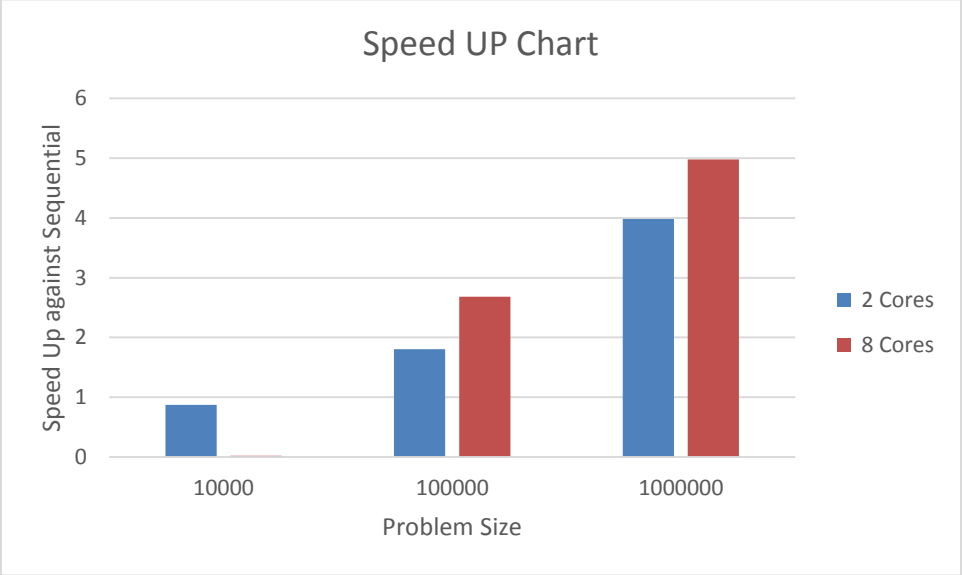
For **OpenMP**, I am utilizing the Selection – Partition Strategy used in the sequential approach above. I am creating one bucket per process. Using Selection-Partition, I create 'k' buckets of equal size, where k is the number of available processes. Each of these buckets are then taken by one process, which sorts it using quicksort algorithm in O(mlogm) time, with m being the size of buckets. Since, blocks are independent of each other, we performed the sort 'In place', therefore further eliminating the time required for copying the data.

2. **Graphs:**

- **Run Time Growth:**



- **Speed Up Plot:**



3. Performance Evaluation:

- **Increase in the size of the problem:**

  From the Graphs, we can observe that the time duration increases very slowly with the increase in problem size. This means that this approach is very scalable and performs very well with increasing problem size. We can get up to 5 fold increase in terms of Speed up with 8 cores. The more cores we increase, the more performance improvement we will get.

- **Increase in the number of cores:**

  From the Speed UP Chart shown above, we can observe that with increasing number of cores we will get more speed up and performance. This is because, as the number of cores increases, more and more number of processes can run in parallel. That means, more buckets will get sorted in parallel and hence the performance gain.

- **Impact of load balance on performance:**

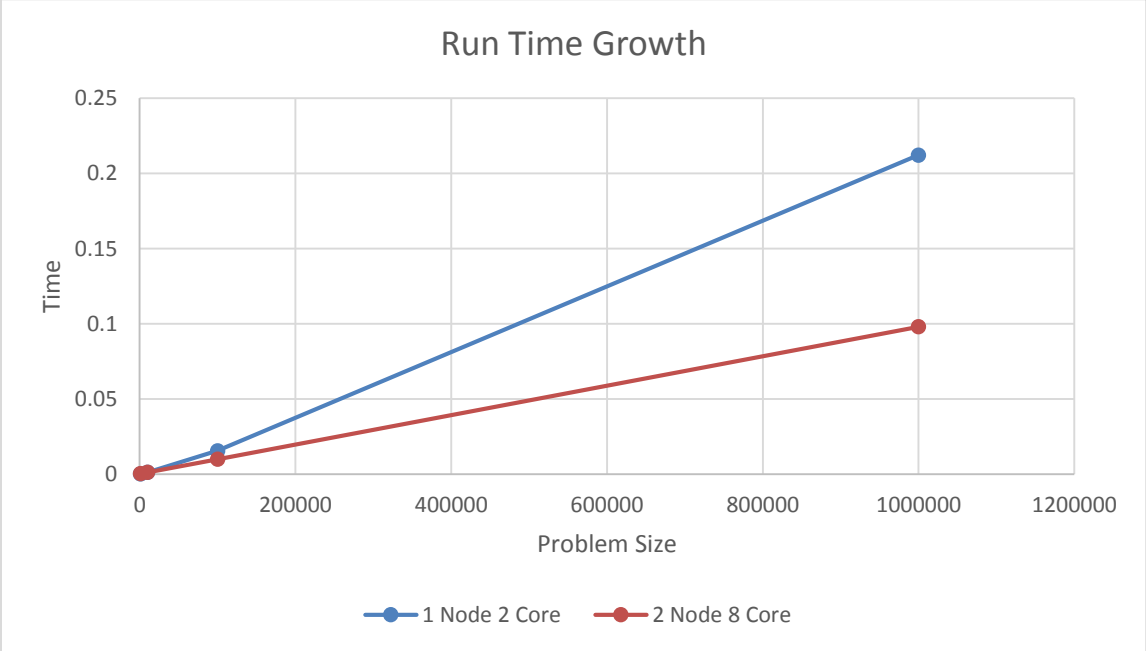
  If the load was not properly balanced, that is, if the buckets were of uneven lengths, the some of the cores would have received more work while others less. In this case, the total execution time will be governed by the slowest task or the task with maximum number of nodes. So even if we have multiple cores which can sort buckets in parallel, if the load is not properly balanced then we will not gain much performance improvement. In the worst case (All load to one node), its performance will be similar to that of Sequential Code.

MPI

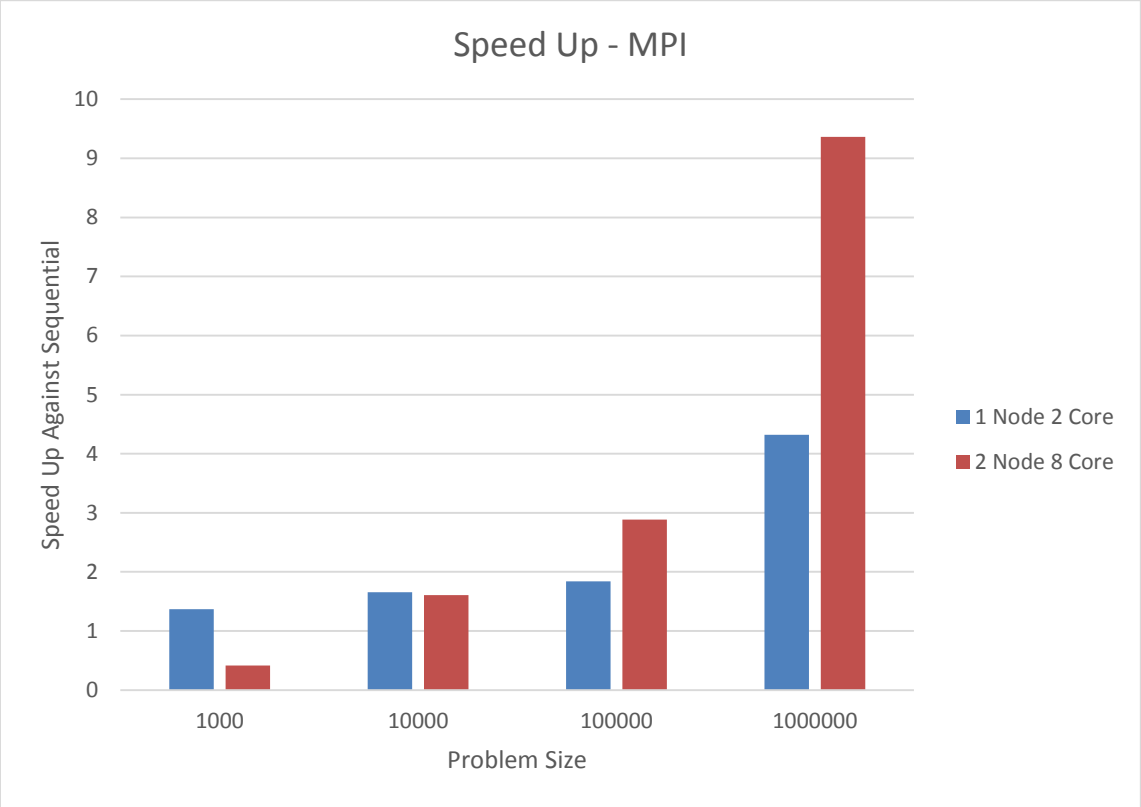
1. **Approach:**

The initial bucket division part is same as that explained above for Sequential and OpenMP. Each process is assigned one bucket. The entire Input data is divided into (Nodes * Cores) buckets. The Selection-Partition part is not parallelized and is worked up on entirely by the master. Once the buckets are created, bucket **'I'** is sent to the process **'I'** according to the rank. Each of these process, then perform the sort in parallel and return back the sorted bucket to the master. This sorted data is then copied back directly to the original array at its proper position (which is obtained from the rank, that is, bucket from process with rank 'I' goes to block 'I').

2. **Graphs:**
    - **Run Time Growth:**



- **Speed Up Plot:**



3. Performance Evaluation:

- **Increase in the size of the problem:**

We can see that we are getting very good performance improvement using MPI as compared to OpenMP. The run time growth is very slow with respect to large input. The speedup is also very impressing. For 2 Node and 8 Cores, that is, for 16 Processes we are getting a speedup of 11 times. This approach is efficient and very scalable.

- **Increase in the number of cores:**

Here we can see very high improvements with increasing number of cores and nodes. In this approach, we are dividing the input data into 'num of processes' buckets. So as either nodes or cores increases, the number of processes will increases so we can perform more sorting in parallel. This is why, increase in this will improve the performance.

- **Impact of load balance on performance:**

This approach faces the same problems with uneven load as that of OpenMP. In some cases, this effect would be much more severe. For example, if the load in uneven and a particular process (in a different node) gets most of work. So, we are losing time both in terms of 'idleness' of other processes but also in terms of network bandwidth. A large load will take a long time to get transferred through and fro. So Load balancing is more severe factor in the case of MPI.

OpenMP + MPI

1. **Approach:**

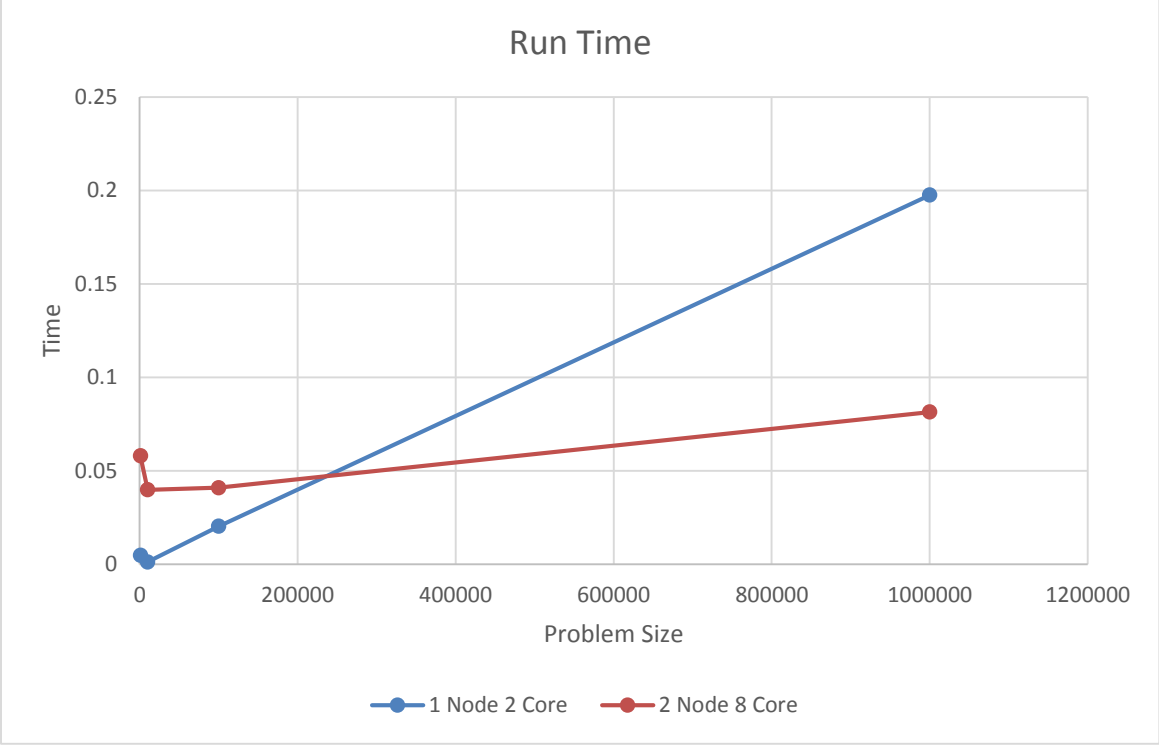
The initial bucket division part is same as that explained above. Each process is assigned one bucket. The entire Input data is divided into (Nodes * Cores) buckets. The Selection-Partition part is not parallelized and is worked up on entirely by the master. The bucket creation is a mix of both OpenMP and MPI approach. MPI will be used between each nodes and within a node the task is divided further by using OpenMP.

I first create a set of large buckets, whose size is determined by (Problem Size/Number of Nodes). These large buckets are created using Selection-Partition for uniform load. These large buckets are then assigned to each nodes using MPI approach (as explained above).

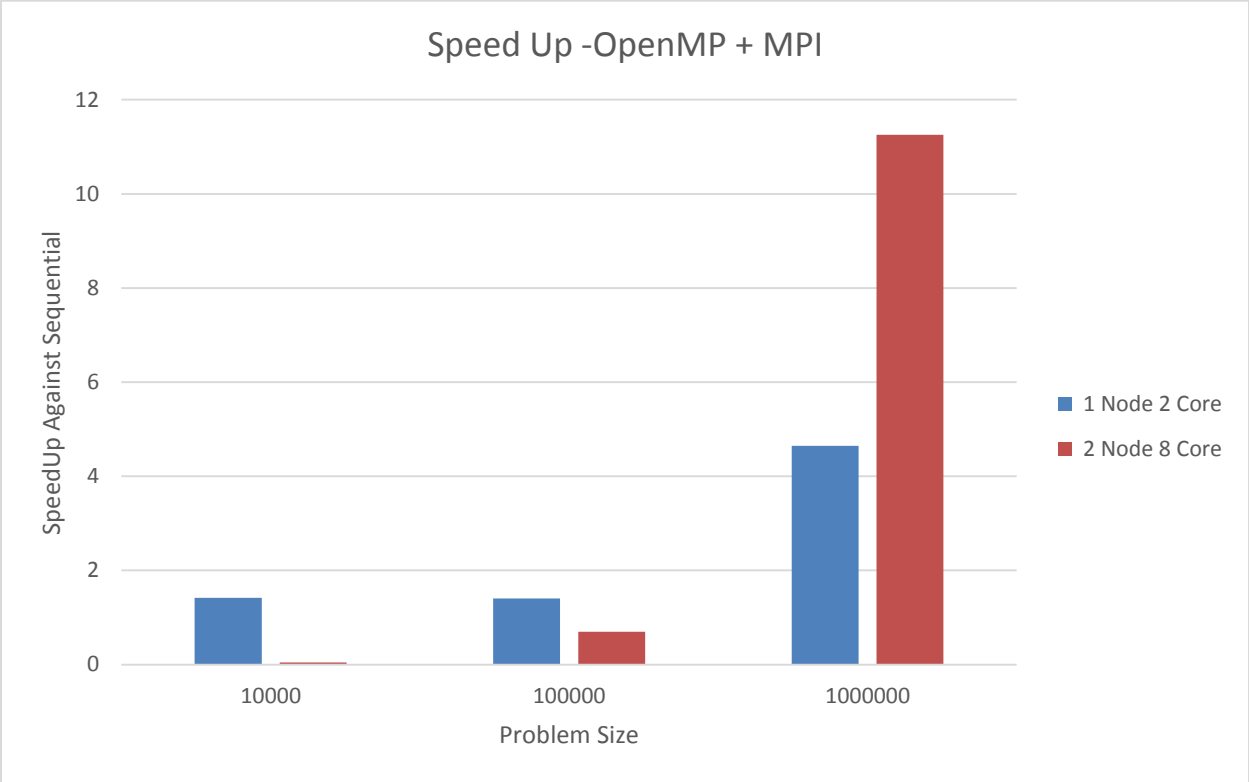
At each nodes, these large buckets are then further divided in to smaller buckets using the same Selection-Partition strategy. These smaller buckets are then assigned to each cores using OpenMP approach. The combined results are then sent back to the master node where it gets consolidated into the original sorted array.

2. **Graphs:**

- **Run Time Growth:**



- **Speed Up Plot:**



3. Performance Evaluation:

- **Increase in the size of the problem:**

  This approach is providing the best performance among all others. The run time growth of this approach is very slow. The large data is effectively divided multiple times and worked up on to achieve high level of parallelism. This approach is fit for large problem size. We can see overhead introduced in case of smaller size, which becomes very negligible as problem size increases.

- **Increase in the number of cores:**

  Due to the nature of this approach, we are able to perform more work in parallel. We can see that with 2 Nodes and 8 Cores, this approach is providing 11 times speedup. In case of MPI, for the same number of nodes and cores, we were getting a speedup of 9 times. Since, the memory is shared for OpenMP processes, unnecessary copying and transmission of data is minimized which improves the performance.

- **Impact of load balance on performance:**

  The impact is similar to the one observed in case of MPI and OpenMP. If the nodes are not assigned same amount of work then only one Process is not staying idle, but all the processes inside a node will remain idle. Also, additional load to one node will increase the data transmission time.

Comparison

The following table presents the comparison of different approaches discussed above.

Program	Number of Cores	Problem Size	Speed Up
OpenMP	1 node 2 task-per-node	100000	1.804188
OpenMP	1 node 2 task-per-node	1000000	3.985497
OpenMP	1 node 8 task-per-node	1000000	4.977541
MPI	1 node 2 task-per-node	100000	1.84266
MPI	1 node 2 task-per-node	1000000	4.32289
MPI	2 node 8 task-per-node	1000000	9.365534
OpenMP_MPI	1 node 2 task-per-node	100000	1.404363
OpenMP_MPI	1 node 2 task-per-node	1000000	4.643748
OpenMP_MPI	2 node 8 task-per-node	1000000	11.25719

We can see that OpenMP + MPI performed best among others.

References

4. **Selection and Partition Algorithm – CSE 531 Slides.**