Algorithms Outline

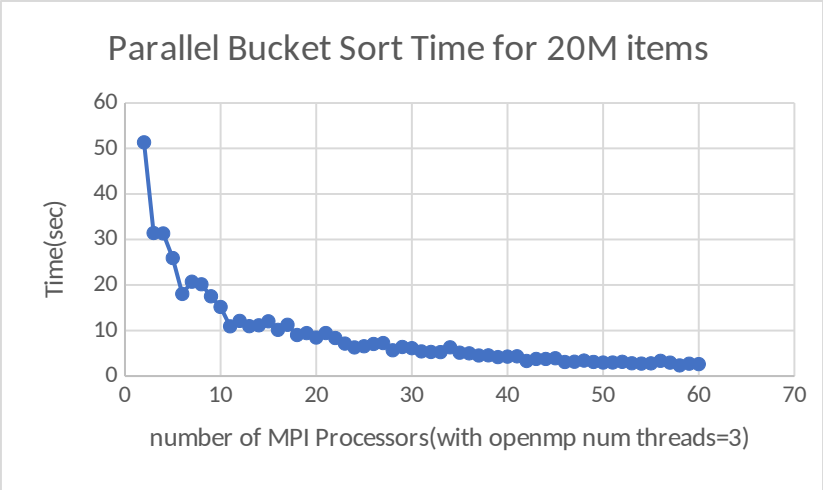
Designed 2 algorithms

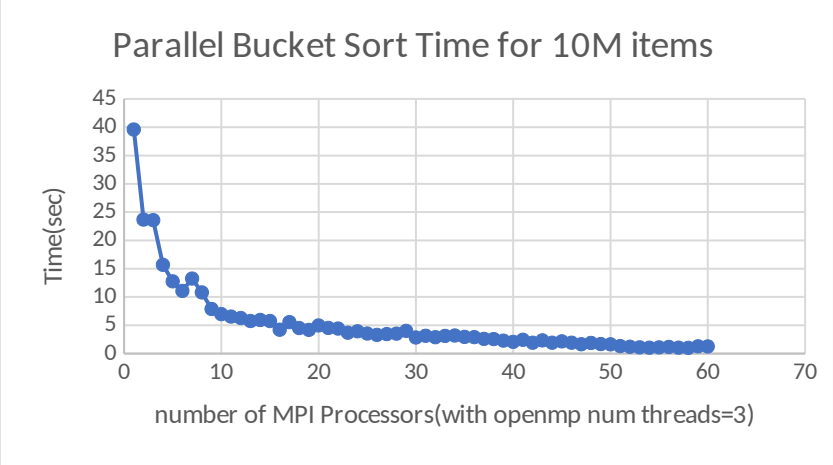
1. Parallel quick sort
    - Step 1. Message passing between processors how many elements each has.
    - Step 2. Processor 1 sends pivot element to each processor (MPI_Isend, MPI_recv)
    - Step 3. Each processor divides itself into 2 parts left (containing items < pivot) and right (containing items >=pivot)
    - Step 4. Message passing to other processors that how many elements smaller than pivot are present with itself
    - Step 5. Processors compute whether they are left, right or pivot processors themselves; and start sending elements between each other swapping elements to get to a state where there are first all elements less than pivot then all with values more than pivot.
    - Step 6. If any processor has all the elements of one side i.e. left or right pivot, it does parallel sorting using openmp.
    - Step 7. Place pivot at the correct place by message passing to correct processor
    - Step 8. Do recursive call on left and right arrays of pivot.
2. Parallel Bucket sort
    - Step 1. Use (openmp) parallel merge sort to on each processor to internally sort each local array.
    - Step 2. Select B-1 Pivots and send this to the processor 0
    - Step 3. Use openmp to sort these B*(B-1) elements.
    - Step 4. Select B-1 splitters
    - Step 5. Each processor starts message passing to send(elements not belonging to itself) and receive(elements that belong to itself).
    - Step 6. Use (openmp) parallel merge sort to on each processor to internally sort each local array.
    - Step 7. Use message passing to rearrange array so that each processor has same number of items that it contains before calling sort.

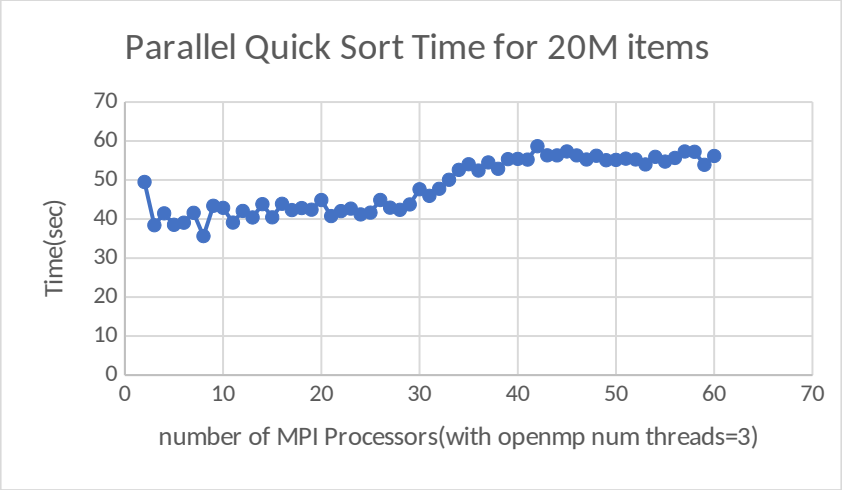
Timing Experiments

All the tables of below graph can be found in time_1.xlsx in the same folder.

Effect of changing number of MPI Processes

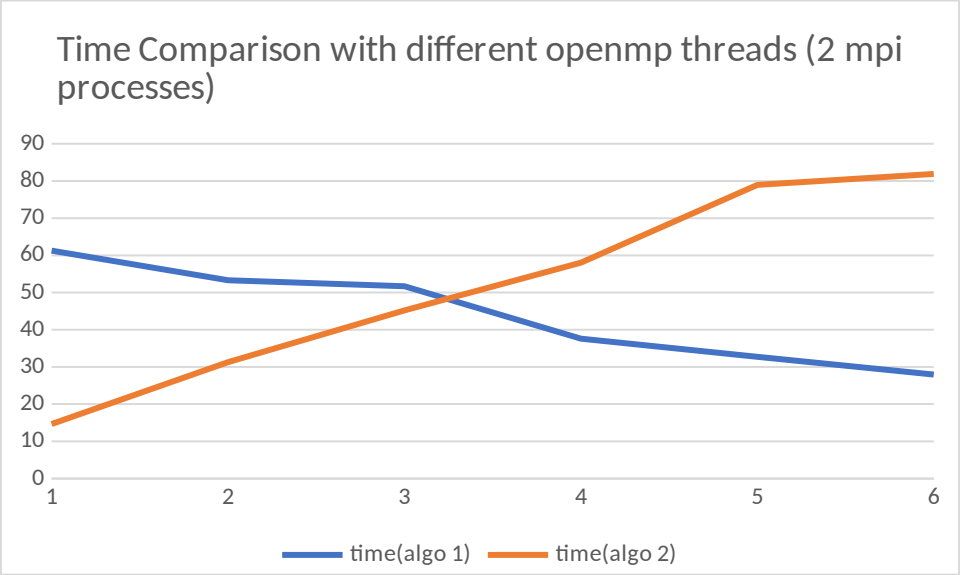


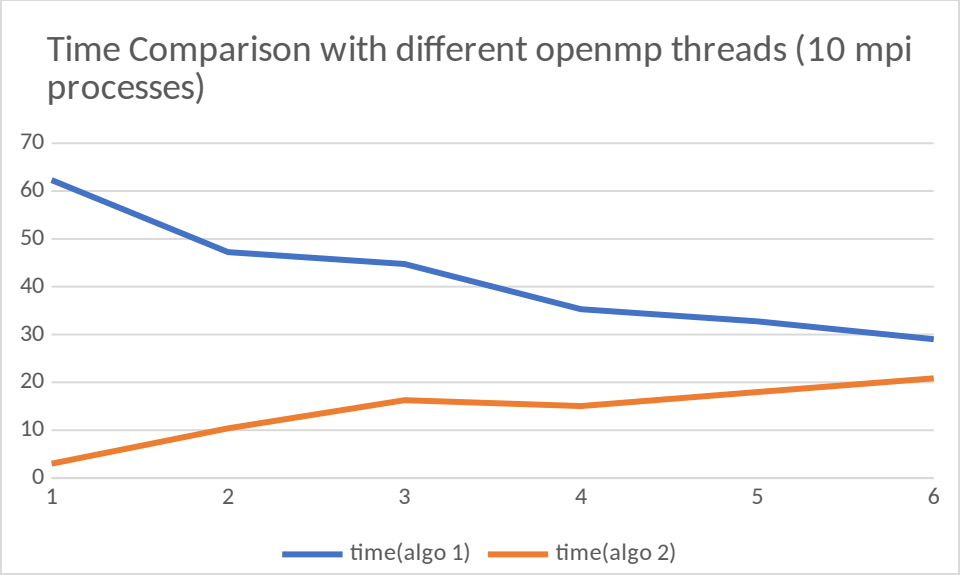


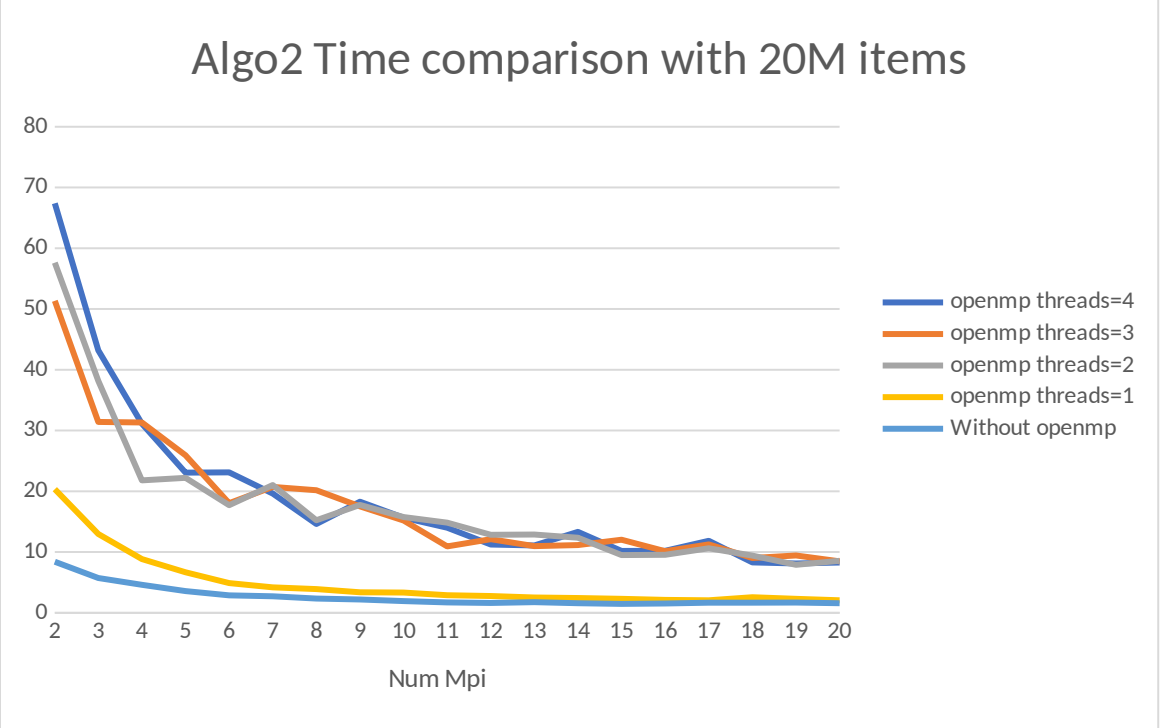


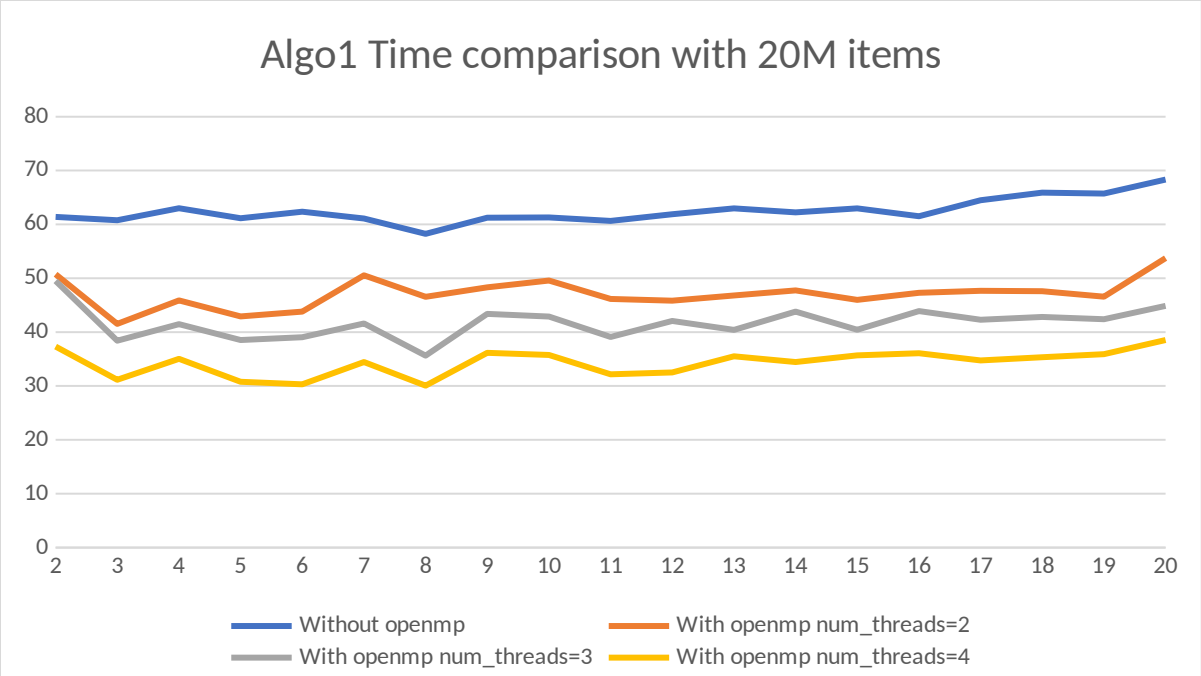
Effect of changing number of openmp num threads

(the following graphs have been improved in the last section. and corresponding excel file is time.xlsx)




Time Comparison with different openmp threads (10 mpi processes)
70
60
50
40
30
20
10
0
1
2
3
4
5
6
time(algo 1)
time(algo 2)


Algo2 Time comparison with 20M items
80
70
60
50
40
30
20
10
0
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Num Mpi
openmp threads=4
openmp threads=3
openmp threads=2
openmp threads=1
Without openmp



Time analysis

From above graphs we can infer that we see an improvement in performance in quick sort with increasing number of openmp threads, whereas the Bucket sort sees improvements with the increase in number of mpi threads(this was corrected in the next section).

The reason of above can be explained by the fact that

1. In quick sort
    a. there is a lot of data shuffling between processors, which takes longer time in case of MPI, so as MPI increases, even though the parallelism increases, it doesn't affect time because of large amount of data transfer between MPI processes.
    b. whereas openmp does data accessing faster with its shared memory architecture, so increasing openmp threads benefits this process.
2. Similarly we see that in bucketized sorting with splitter
    a. Data transfer happens only in the beginning and the end, which helps it in keeping the amount data transfer to low so increasing MPI processes benefits the algorithm
    b. It brings all the data of each processor to itself in the beginning itself, reducing the number of times different MPI processes need to communicate,
    c. Also openmp has faster shared memory access among threads on the same node this makes internal merge sort in a node faster, but this does not seem to help our algorithm. May be we need to try restricting the splitting at lower lengths of array which would help us in taking better advantage of openmp(see next section).

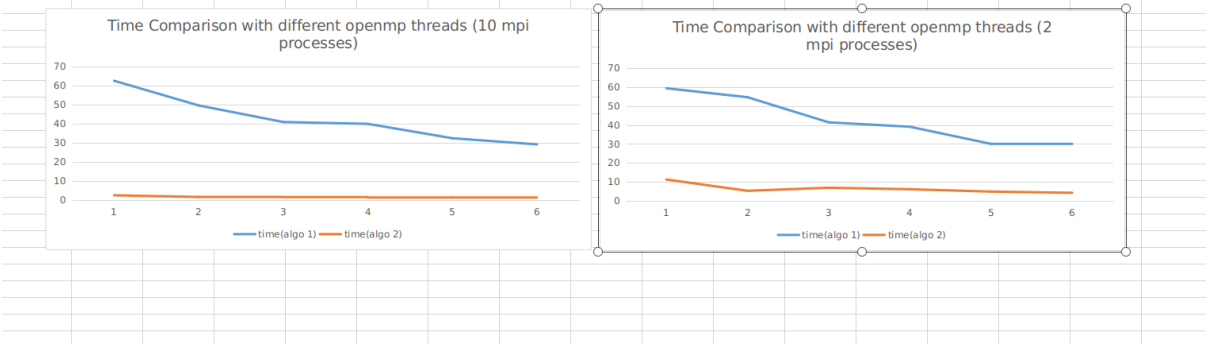
Further Improvements Made in algorithms

Upon making the above changes of restricting the splitting at lower lengths of array,and further

analysis and taking an extra day of assignment, we see a significant improvement in the time taken and increasing the openmp threads is helping the algorithm in reducing time which was not seen earlier.

We see that there increasing openmp threads reduces the time in algo 2 as well(can be seenin the below graphs;time.xlsx).

Mpi_procs	openmp nu	time(algo 1	time(algo 2	Mpi_procs	openmp nu	time(algo 1	time(algo 2)
10	1	62.595	2.381	2	1	59.407	11.169
10	2	49.607	1.502	2	2	54.67	5.191
10	3	40.903	1.433	2	3	41.389	6.816
10	4	39.938	1.214	2	4	39.064	6.018
10	5	32.366	1.201	2	5	30.003	4.773
10	6	29.145	1.129	2	6	29.597	4.14



Amount of time taken has also been significantly reduced. earlier it was taking too much time as can be seen in below table.

Mpi_procs	openmp numthreads	Before change	After change
2	1	14.634	11.169
2	2	31.27	5.191
2	3	45.199	6.816
2	4	58.033	6.018
2	5	78.928	4.773
2	6	81.897	4.14

The other graph of openmp comparison has also improved and shows the relation between the openmp threads and execution time.

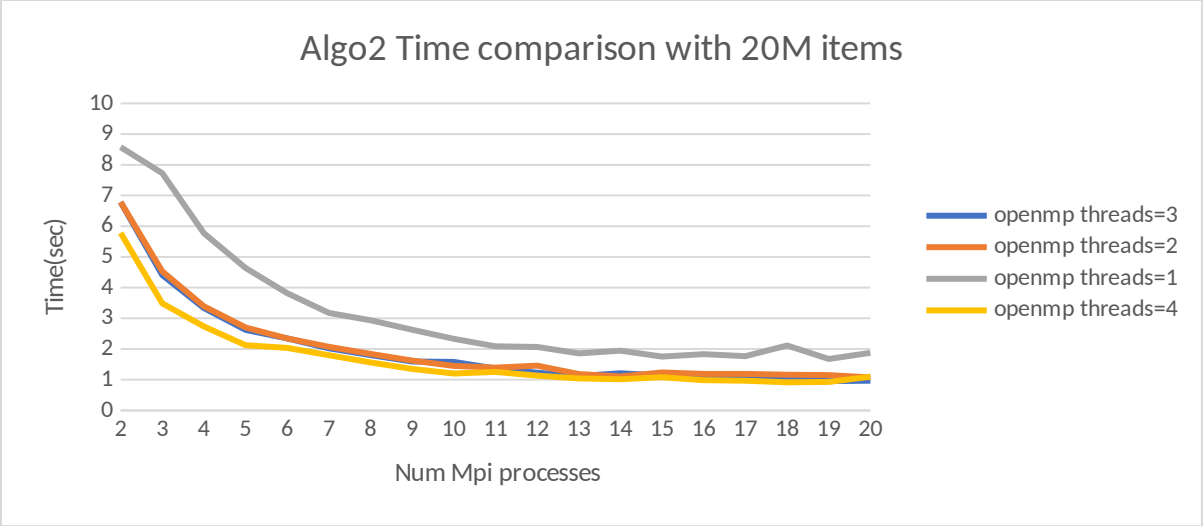


Table for same is shown below.

Num Mpi	openmp threads=4	openmp threads=3	openmp threads=2	openmp threads=1
2	5.782	6.784	6.784	8.572
3	3.488	4.405	4.529	7.727
4	2.735	3.334	3.391	5.771
5	2.122	2.622	2.701	4.64
6	2.036	2.344	2.344	3.816
7	1.794	2.023	2.07	3.175
8	1.563	1.801	1.841	2.933
9	1.348	1.591	1.621	2.627
10	1.202	1.579	1.446	2.329
11	1.256	1.364	1.389	2.089
12	1.132	1.228	1.458	2.065
13	1.041	1.12	1.184	1.856
14	1.015	1.209	1.105	1.943
15	1.082	1.13	1.238	1.753
16	0.986	1.059	1.187	1.829
17	0.965	1.145	1.188	1.767
18	0.915	1.031	1.161	2.113
19	0.929	0.953	1.147	1.674
20	1.095	0.969	1.073	1.873