MPI Implementation

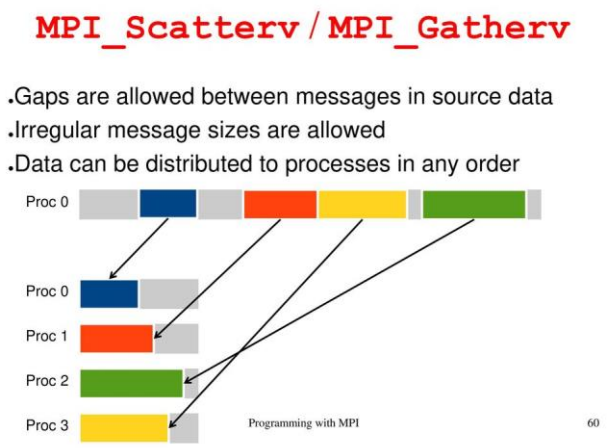
The algorithm implemented in the MPI environment is a relatively simple one. Essentially, it is the "central" node of the cluster, with rank = 0, which takes over to divide the elements of the array to be sorted into the right buckets, having previously calculated the range of values (range) and the variable dividing_num. Each bucket is then allocated to its assigned node, according to its rank, and this node takes over to sort the elements of the bucket. Finally, the "central" node takes over again to collect the elements of the different sorted buckets and bring them together, forming the final sorted array. However, the point to focus on in the MPI implementation is the communication and transfer of buckets between the individual nodes and the "central" node. Specifically, the sending of buckets from the "central" node to the other nodes was done using the "MPI_Scatterv" command, as shown below:

```
MPI_Scatterv(&(*init_mem), sizes , displs_n, MPI_FLOAT, &local_array[0]
, my_size, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

After first sending the bucket size (my_size) from the "central" node to each individual node in the cluster, so that the latter knows the size of the message it will receive.

```
MPI_Send(&size_n,1,MPI_INT, i ,tag, MPI_COMM_WORLD);
MPI_Recv(&my_size, 1 , MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
```

"MPI_Scatterv" is an extremely useful command, as it allows variable size arrays to be sent between nodes. This is made possible by combining the definitions of displs_n (here displs_n) and sizebuf (here sizes) which essentially uniquely define a range in a table. The first argument of these defines the distance, in memory, of the address specified in sendbuf (here &(*init_mem)), while the second defines the size of the sub-array to be sent. Below, the function of "MPI_Scatterv" is briefly presented, to better understand the function and its importance for our algorithm.



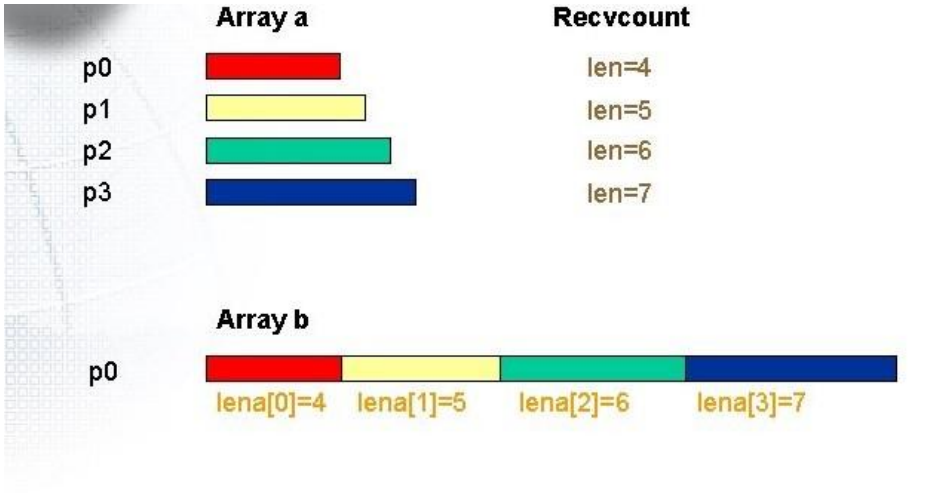
Essentially, "MPI_Scatterv" was chosen over "MPI_Scatter" or the combination of "MPI_Send" and "MPI_Recv", both because of its speed and because of its ability to transmit buckets directly, without any intermediate resizing, to the various nodes. Indeed, as mentioned above, the size of the buckets varies according to the elements of the table to be sorted. Then, after knowing its bucket, each node sorts it, as shown below:

```
sort(local_array.begin(), local_array.end());
```

Finally, once the classification is done, the "central" node takes over to collect the data from the different buckets and form the single final sorted table. This sorting is done using the "MPI_Gatherv" command, as shown below:

```
MPI_Gatherv(&local_array[0], my_size_n , MPI_FLOAT, &sort_array[0] ,
array_sizes , displs , MPI_FLOAT, 0, MPI_COMM_WORLD);
```

The syntax of this command is similar to that of "MPI_Scatterv" and allows us to gather individual buckets, which are of variable size, into the "main" node. This is its main advantage over similar methods ("MPI_Scatter" and "MPI_Recv") and is the main reason why it was used in our code. Below is a diagram of how it works:




```
MPI_Gatherv(a,len,MPI_INT,b,lena,MPI_INT,0,MPI_COMM_WORLD)
```

Finally, we provide a snapshot of the execution of the MPI algorithm for a 50-element matrix for 8 and 16 nodes respectively:

```
Sorted array is
197.239 448.437 546.291 802.965 929.609 1066.49 1096.5 1283.81 1301.86 1495.58 2089.54 2425.19 2756.62 2883.53 3066.27 3104.83 3150.66 3299.03 3352.
29 3393.07 3443.58 4040.44 4405.56 4437.22 5018.82 5122.97 5194.38 5479.61 5579.7 5712.41 5745.23 5958.51 6160.84 6365.75 6509.85 6543.73 6743.3 692
3.96 7090.02 7209.49 7698.35 7845.13 8555.65 8778.63 8828.17 8876.67 8895.38 8968.27 9242.84 9786.72
Total time for Scatterv is 0.001144 sec
Total time for Gatherv is 0.000227 sec
Total time for Receive is 0.013843 sec
Total time for Send is 0.000123 sec
Total Communication cost: 0.015337 sec
Total time for execution is 0.435313 sec
[ioanlili@dioptis1 erg_21_cpp]$ mpirun -machinefile m3 -np 16 bucket_parallel_mpi4 50
Sorted array is
197.239 448.437 546.291 802.965 929.609 1066.49 1096.5 1283.81 1301.86 1495.58 2089.54 2425.19 2756.62 2883.53 3066.27 3104.83 3150.66 3299.03 3352.
29 3393.07 3443.58 4040.44 4405.56 4437.22 5018.82 5122.97 5194.38 5479.61 5579.7 5712.41 5745.23 5958.51 6160.84 6365.75 6509.85 6543.73 6743.3 692
3.96 7090.02 7209.49 7698.35 7845.13 8555.65 8778.63 8828.17 8876.67 8895.38 8968.27 9242.84 9786.72
Total time for Scatterv is 0.003696 sec
Total time for Gatherv is 0.000591 sec
Total time for Receive is 0.010465 sec
Total time for Send is 0.000237 sec
Total Communication cost: 0.014989 sec
Total time for execution is 0.483990 sec
```

As can be seen, the communication cost of sending buckets, given that the table size is quite small, is minimal. However, as we will show in a later section, this cost increases with the size of the buckets.

Results Presentation

Here we present both the tables with the requested execution times, speed and efficiency, and the appropriate graphs and bar charts to make the conclusions drawn more obvious. These tables are as follows:

- Total Execution Time Table

Total Execution Time								
Array Length	Serial	For 1 Bucket	For 2 Buckets	For 4 Buckets	For 8 Buckets	For 16 Buckets	For 24 Buckets	For 32 Buckets
131072	0,099687	0,086153	0,070883	0,046676	0,036995	0,13375	0,242189	0,435536
262144	0,212822	0,180459	0,144782	0,095283	0,072978	0,069827	0,16041	0,152399
524288	0,430468	0,375334	0,303754	0,196519	0,147966	0,130089	0,434527	0,332069
1048576	0,965821	0,787011	0,625607	0,404049	0,299986	0,253963	0,264111	0,281873

- Speedup Table

Speedup							
Array Length	1 Bucket	2 Buckets	4 Buckets	8 Buckets	16 Buckets	24 Buckets	32 Buckets
131072	1,157092614	1,406359776	2,135722855	2,694607379	0,745323364	0,411608289	0,228883491
262144	1,179337135	1,469947922	2,233577868	2,916248732	3,047846821	1,326737735	1,39647898
524288	1,146893167	1,417159939	2,190465044	2,909235906	3,309026897	0,990658808	1,296320945
1048576	1,227201399	1,543814248	2,39035612	3,219553579	3,802998862	3,656875329	3,426440276

- Efficiency Table

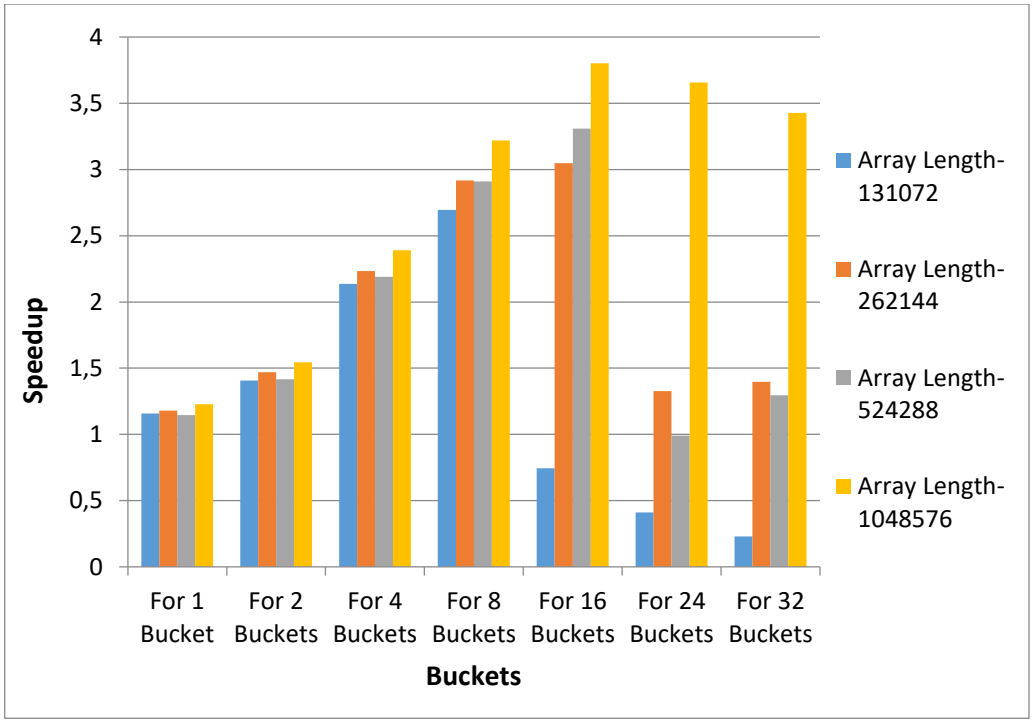
Efficiency							
Array Length	1 Bucket	2 Buckets	4 Buckets	8 Buckets	16 Buckets	24 Buckets	32 Buckets
131072	1,157092614	0,703179888	0,533930714	0,336825922	0,04658271	0,017150345	0,007152609
262144	1,179337135	0,734973961	0,558394467	0,364531092	0,190490426	0,055280739	0,043639968
524288	1,146893167	0,708579969	0,547616261	0,363654488	0,206814181	0,04127745	0,04051003
1048576	1,227201399	0,771907124	0,59758903	0,402444197	0,237687429	0,152369805	0,107076259

In addition, the communication times (costs) between the network nodes were also measured for each number of available buckets. However, the large number of measurements made does not allow us to include all these tables. Therefore, we present only the following representative table of communication times for 8 buckets:

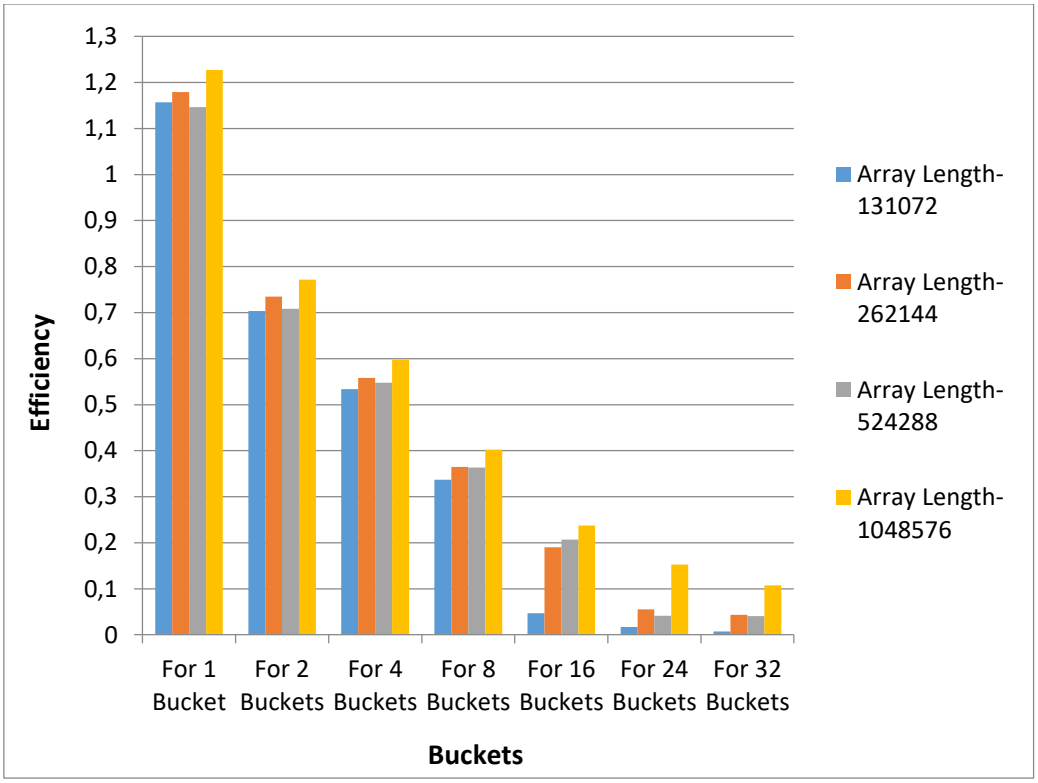
For 8 Buckets						
Array Length	Time for Scatterv	Time for Gatherv	Time for Receive	Time for Send	Total Communication Cost	Total Execution Time
131072	0,004641	0,007018	0,000004	0,000235	0,011898	0,036995
262144	0,010513	0,01241	0,000006	0,000233	0,023162	0,072978
524288	0,020697	0,024322	0,000005	0,000246	0,04527	0,147966
1048576	0,038133	0,04483	0,000005	0,000247	0,083215	0,299986

Essentially, as mentioned above, time measurements were performed for all communication functions used in our code. Finally, we list all the graphs and bar charts associated with the metrics mentioned above. These are

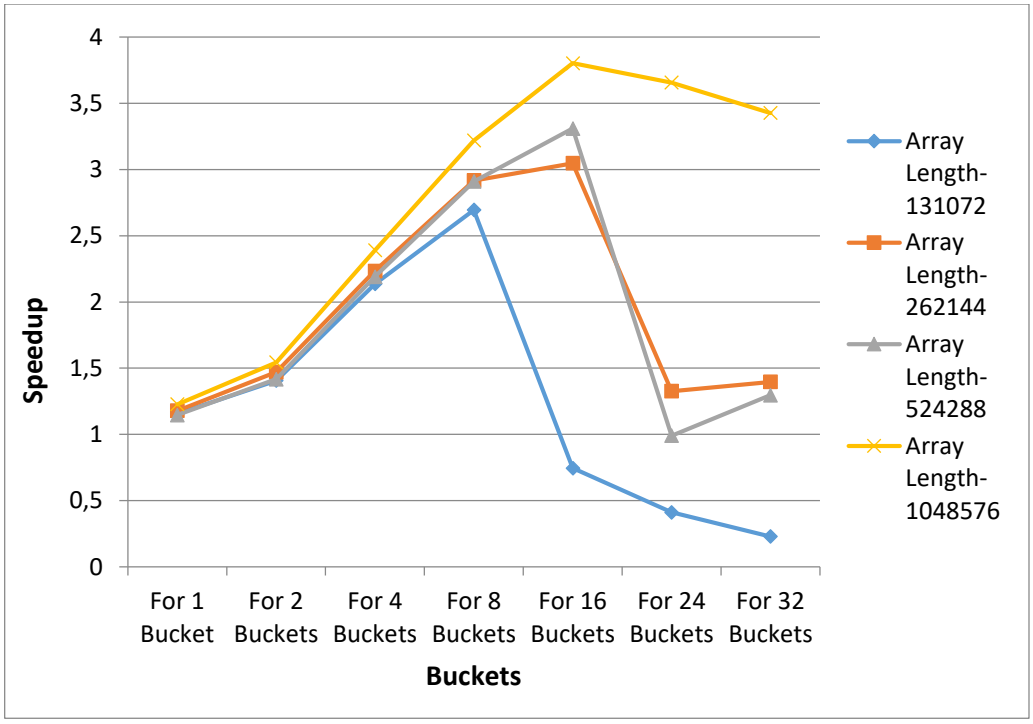
- Speedup – Bucket Number bar graph, depending on table size



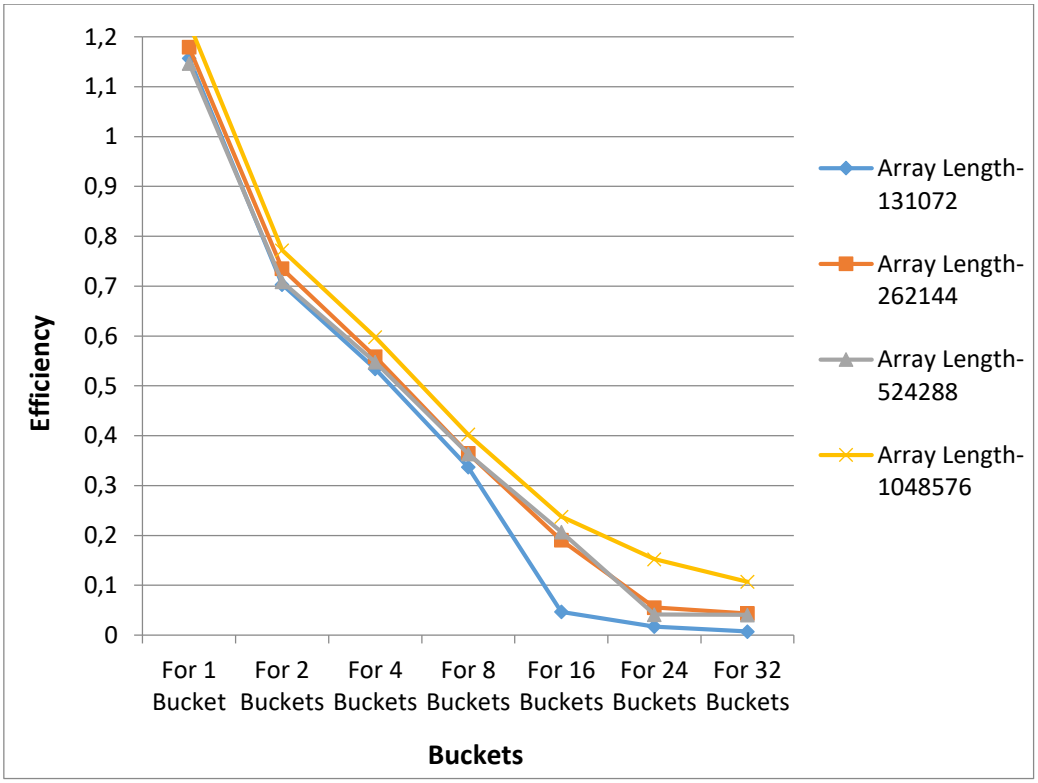
- Efficiency – Bucket Number bar graph, depending on table size



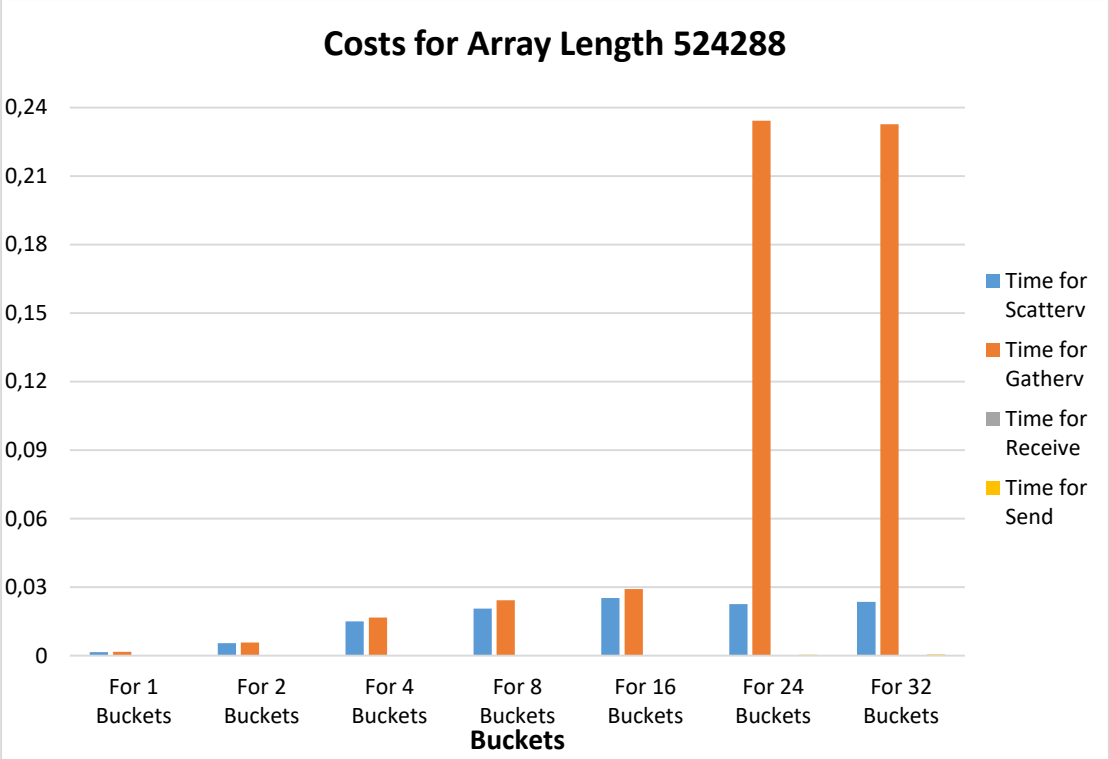
- Speedup – Bucket Number diagram, depending on table size



- Efficiency – Bucket Number diagram, depending on table size



Finally, we also provide a bar graph showing the time required to execute the MPI communication functions implemented in our code, for a table size of 524288 elements:



As can be seen, the communication functions that take the longest to execute are the "MPI_Scatterv" and "MPI_Gatherv" functions. Moreover, this time increases considerably when the number of available buckets increases. This is perfectly reasonable, since these functions are related to sending and receiving buckets from individual nodes.