Bilkent University
Department of Computer Engineering

CS 426
Project 2

Nisa Yılmaz
22001849

Description of Implementation

The implementation of sequential quicksort is trivial. I used Hoare's partition algorithm for partitioning the data.

For the parallel implementation, the master processor reads the text file and sends the total number of integers using MPI_Bcast collective communication to all processors. After receiving the total number, each processor divides this by the number of processors to find its workload and allocates an array. Then using MPI_Scatter the master processor sends all processors their portion of the main array. After receiving their portion, all processors use the sequential quicksort to sort their local arrays.

After this part, there is a while loop that executes as long as the total number of the processors in the communicator is greater than 1. This recursively sorts the array until the sub-cube cannot be divided anymore. In the while loop all of the processors find their median and perform a MPI_Allgather operation to send their median to all other processors as well as receive theirs. All of the medians are collected in an array in all processors, then all processors sort their medians array and find the median of medians to be used as the pivot element.

Then the mask is found as 2 raised to the power dimension - 1. Each processor finds its partner by using their id XOR mask and their masked id using their id AND mask. If their id starts with 0 (they are in the lower subcube) their masked id is 0. If this is the case the processor finds the values that are greater than or equal to the median of medians in their local array and sends this array to their partner processor. Else, the processor finds the values smaller than the median of medians and sends this part. After receiving the array from their partner, all processors merge the incoming array with their local array using the merge sort merge step. This can be done since both of the arrays are sorted. Then the subcube is split into two by using masked id values. Also the dimension is decreased.

When the while loop terminates, there is 1 processor in each communicator with a sorted local array. These local arrays are then collected in the master processor using MPI_Gatherv. The gathered array is sorted since each array has a sorted local array and the gather operation is done based on the ids (gathers starting from the smallest id).

Tests

I tested both implementations using 3 different values for small, medium and large sized inputs. For each input I used 1, 2, 3, 4, 5, and 6 dimensional hypercubes. For generating random arrays I created a python script. The resulting data can be seen below.

	Sequential	1 Dim	2 Dim	3 Dim	4 Dim	5 Dim	6 Dim
16000	0.001957	0.001808	0.001081	0.020139	0.184479	1.010838	3.312429
800000	0.129395	0.082565	0.113465	0.202976	0.668723	1.688186	2.980515
30000000	5.427797	2.954824	1.793322	1.361102	2.588937	6.927537	19.4494

Table 1. Test Data

The below graphs show the runtime for input sizes (Figure 1, Figure 2, Figure 3) as well as the whole Table 1. For the graphs that show the runtime for given input, 5 and 6 dimensions are left out since they exceed the sequential runtime to a great extent and make it harder to interpret the graph.

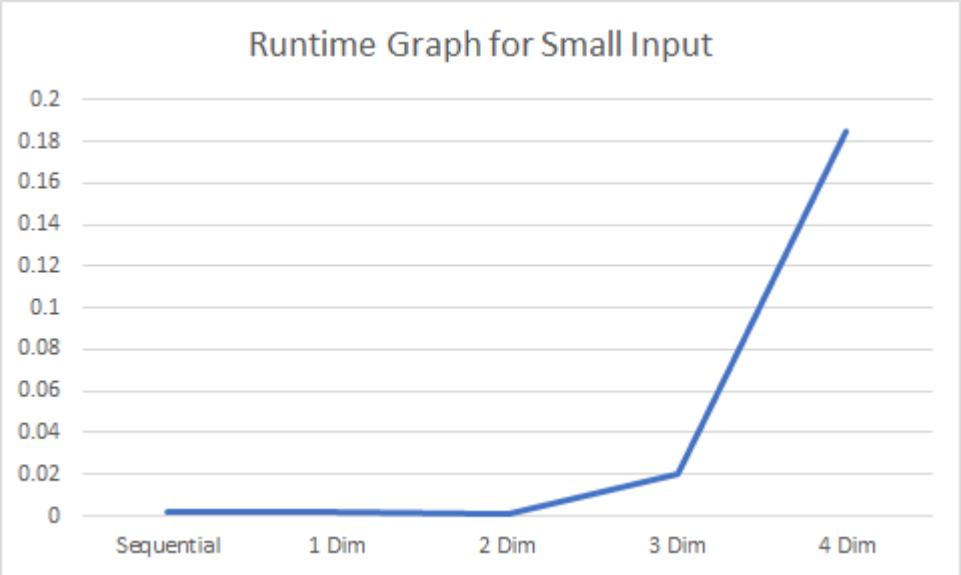

Figure 1. Runtime Graph for Small Input

From the graph above, It can be seen that for small input sizes parallel implementation does not provide any significant speedup since the sequential quicksort itself is very fast. Even Though from Table 1 it can be observed that there is a slight speedup with 1 dimension (2 processors) and 2 dimension (4 processors) these are very small speedups. Also, after 2

dimensions the runtime exceeds the sequential runtime because of the overheads of communication.

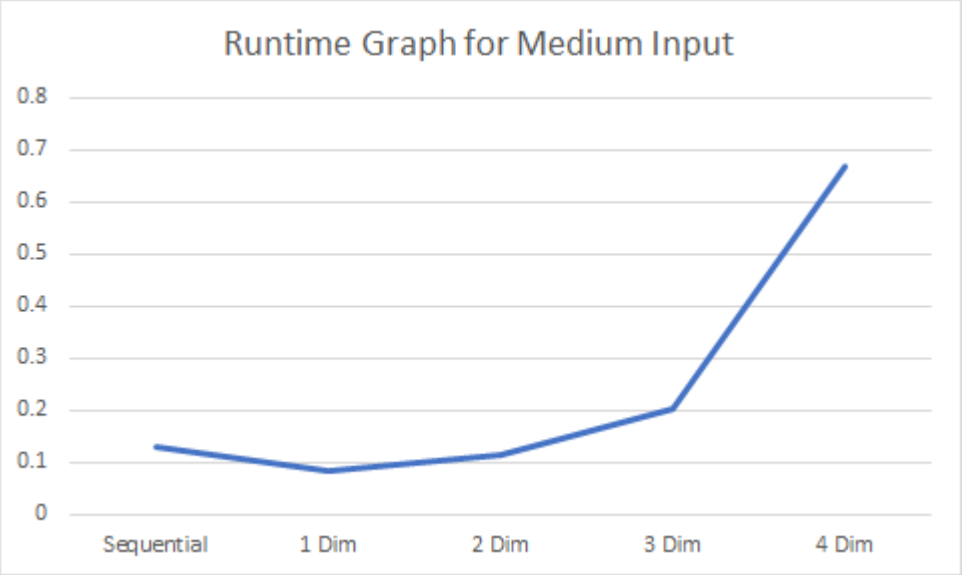

Figure 2. Runtime Graph for Medium Input

For medium input size, a more significant speedup can be seen with parallel implementation. Both 1 dimension and 2 dimensions result in a speedup, but the runtime for 2 dimensions is greater than 1 dimension. Anything greater than 2 dimensions exceeds the sequential runtime, with the same reason mentioned above for small input.

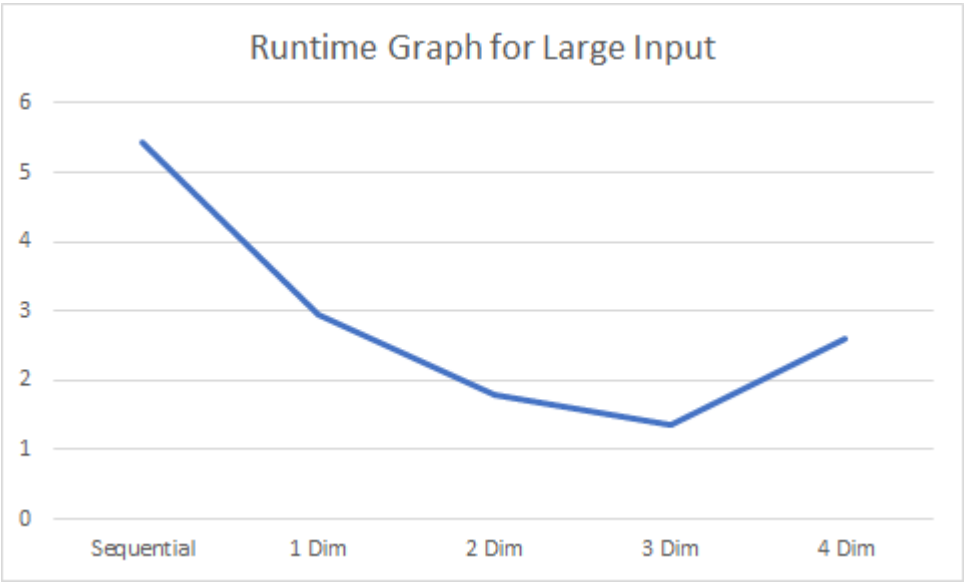

Figure 3. Runtime Graph for Large Input

The most significant speedup can be seen with large input. 1 dimension provides a speedup of almost 2. The runtime decreases until 4 dimensions. It can be seen from the graph that the ideal parallelization for this input size is accomplished with 8 processors. The runtime with 4 dimensions is still smaller than sequential, but this value is not the optimal so it still has some communication overheads increasing the runtime.

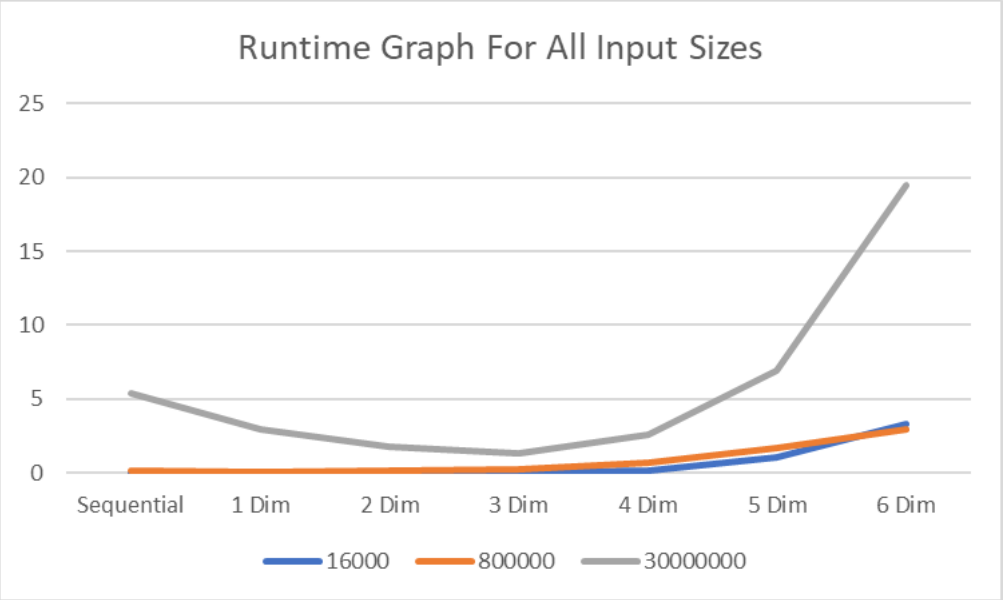

Figure 4. Runtime Graph for All Input Sizes

The graph for the whole Table 1 shows that as the number of processors increase beyond 16 processors, there is a significant overhead for all input sizes. This is because the algorithm for quicksort on hypercube has a lot of communication operations, and thus a high communication overhead. Also we can observe that the most speedup is observed with large input size.