**Mini Project**

Name: Poonam Kisan Salbande

Roll No: 20121011

Title: Evaluate Performance enhancement of parallel Quick Sort Algorithm using MPI.

Code:-

#include <mpi.h>

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void parallelQuickSort(int arr[], int n, int rank, int size) {

if (n <= 1) {

return;

}

int mid = n / 2;

int pivot = arr[mid];

// Scatter the array to the processes

int local\_arr[mid + 1];

MPI\_Scatter(arr, mid + 1, MPI\_INT, local\_arr, mid + 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Sort the local array

quickSort(local\_arr, 0, mid - 1);

// Gather the sorted subarrays

int global\_arr[n];

MPI\_Gather(local\_arr, mid + 1, MPI\_INT, global\_arr, mid + 1, MPI\_INT, 0,

MPI\_COMM\_WORLD);

// If this is the root process, print the sorted array

if (rank == 0) {

for (int i = 0; i < n; i++) {

printf("%d ", global\_arr[i]);

}

printf("\n");

}

}

int main(int argc, char\*\* argv) {

int n;

if (argc < 2) {

printf("Usage: %s <array\_size>\n", argv[0]);

return 1;

}

n = atoi(argv[1]);

// Initialize MPI

MPI\_Init(&argc, &argv);

// Get the number of processes

int size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Get the rank of the current process

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

// Create an array

int arr[n];

for (int i = 0; i < n; i++) {

arr[i] = rand() % 100;

}

// Sort the array in parallel

parallelQuickSort(arr, n, rank, size);

// Finalize MPI

MPI\_Finalize();

return 0;

}

Output:

