Project Report

1) Brief Description

The project was about sorting an array of integers. Implemented using merge sort.

Sequential Version: The implementation is composed of two methods. The first on is responsible for dividing the array into two parts till we reach the base case.

```
void mergeSort(int l, int r) {
    if (l == r) {
        return;
    }
    int mid = (l + r) / 2;
    mergeSort(l, mid);
    mergeSort(mid + 1, r);
    int left[mid - l + 1], right[r - mid];
    for (int i = l; i <= mid; i++)
        left[i - l] = a[i];
    for (int i = mid + 1; i <= r; i++)
        right[i - (mid + 1)] = a[i];
    merge(l, r, left, right);
}
```

The other method is responsible for merging the two given sorted arrays into one sorted array.

```
void merge(int start, int end, int *left, int *right) {
    int l = 0, r = 0;
    int mid = (start + end) / 2;
    int lenLeft = mid - start + 1;
    int lenRight = end - mid;
    for (int i = start; i <= end; i++) {
        if (r == lenRight || (l < lenLeft && left[l] <= right[r])) {
            a[i] = left[l++];
        } else
            a[i] = right[r++];
    }
}
```

Open MPI Versions

- Send and Receive : For each process, we send to it from its parent the left and right pointer to the array it is responsible for sorting. The process then divides the array and then send the two new left and right pointers to its child processes. Then after the children are done with the sorting it receives back the two sorted arrays and merges them into one sorted array and send it back to its parent and so on ….

```
if (rank == 0) {
    l = 0;
    r = n - 1;
} else {
    MPI_Recv(&l, 1, MPI_INT, parent, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&r, 1, MPI_INT, parent, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
if (l == r || (2 * rank + 2 >= world_size)) {
    mergeSort(l, r);
} else {
    int mid = (l + r) / 2, mid2 = mid + 1;
    int left = rank * 2 + 1, right = left + 1;
    MPI_Send(&l, 1, MPI_INT, left, 0, MPI_COMM_WORLD);
    MPI_Send(&mid, 1, MPI_INT, left, 0, MPI_COMM_WORLD);
    MPI_Send(&mid2, 1, MPI_INT, right, 0, MPI_COMM_WORLD);
    MPI_Send(&r, 1, MPI_INT, right, 0, MPI_COMM_WORLD);
    int n1 = mid - l + 1, n2 = r - mid2 + 1;
    int b[n1], c[n2];
    for (int i = 0; i < n1; i++)
        MPI_Recv(&b[i], 1, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (int i = 0; i < n2; i++)
        MPI_Recv(&c[i], 1, MPI_INT, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    merge(l, r, b, c);
}
if (rank != 0)
    for (int i = l; i <= r; i++)
        MPI_Send(&a[i], 1, MPI_INT, parent, 0, MPI_COMM_WORLD);
else
    for (int i = l; i <= r; i++)
        printf("%d ", a[i]);
```

- BroadCast and Reduce : In this version, we used the broadcast to have the root only read the array from the file and then broadcast it to all processes, and then we continued to use send and receive normally since the reduce does not make sense because we do not have an mpi sort function that can automatically be used.

```
if (rank == 0) {
    FILE *pFile;
    pFile = fopen("test1.txt", "r");
    fscanf(pFile, "%d", &n);
    for (int i = 0; i < n; i++)
        fscanf(pFile, "%d", &a[i]);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
for (int i = 0; i < n; i++)
    MPI_Bcast(&a[i], 1, MPI_INT, 0, MPI_COMM_WORLD);
if (rank % 2 != 0 && rank + 1 >= world_size) {
    MPI_Finalize();
    return 0;
}
if (rank == 0) {
    l = 0;
    r = n - 1;
} else {
    MPI_Recv(&l, 1, MPI_INT, parent, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&r, 1, MPI_INT, parent, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

if (l == r || (2 * rank + 2 >= world_size)) {
    mergeSort(l, r);
} else {
    int mid = (l + r) / 2, mid2 = mid + 1;
    int left = rank * 2 + 1, right = left + 1;
    MPI_Send(&l, 1, MPI_INT, left, 0, MPI_COMM_WORLD);
    MPI_Send(&mid, 1, MPI_INT, left, 0, MPI_COMM_WORLD);
    MPI_Send(&mid2, 1, MPI_INT, right, 0, MPI_COMM_WORLD);
    MPI_Send(&r, 1, MPI_INT, right, 0, MPI_COMM_WORLD);

    int n1 = mid - l + 1, n2 = r - mid2 + 1;
    int b[n1], c[n2];
    for (int i = 0; i < n1; i++)
        MPI_Recv(&b[i], 1, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (int i = 0; i < n2; i++)
        MPI_Recv(&c[i], 1, MPI_INT, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    merge(l, r, b, c);
}
if (rank != 0)
    for (int i = l; i <= r; i++)
        MPI_Send(&a[i], 1, MPI_INT, parent, 0, MPI_COMM_WORLD);
else
    for (int i = l; i <= r; i++)
        printf("%d ", a[i]);
```

- Scatter and Gather: In this milestone we divide the array equally to parts equaivilent to the number of processors in the current world and then scatter theses parts from the root to all other processes. Each process after receiving its part it sort it using sequential milestone. At the root after all parts are sorted we merge these sorted pars together forming the whole sorted array.

```
int n;
pFile = fopen("test1.txt", "r");
fscanf(pFile, "%d", &n);
if (rank == 0) {
    for (int i = 0; i < n; i++)
        fscanf(pFile, "%d", &a[i]);
}
int size = n / world_size;
int receive[size];
MPI_Scatter(a, size, MPI_INT, receive, size, MPI_INT, 0, MPI_COMM_WORLD);
mergeSort(0, size - 1, receive);
MPI_Gather(receive, size, MPI_INT, a, size, MPI_INT, 0, MPI_COMM_WORLD);
if (rank == 0) {
    int l = n - n % world_size, r = n - 1;
    if (l < r) mergeSort(l, r, a);
    for (int i = 0; i < world_size; i++) {
        int m = size * (i + 1);
        int left[m];
        int mm = size;
        if (i == world_size - 1) mm = n % world_size;
        if (mm == 0) break;
        int right[mm];
        for (int j = 0; j < m; j++)
            left[j] = a[j];
        for (int j = m; j < m + mm; j++)
            right[j - m] = a[j];
        merge(0, m + mm - 1, left, right, m, mm, a);
    }
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
}
```

2) Brief Summary of how the work was split:

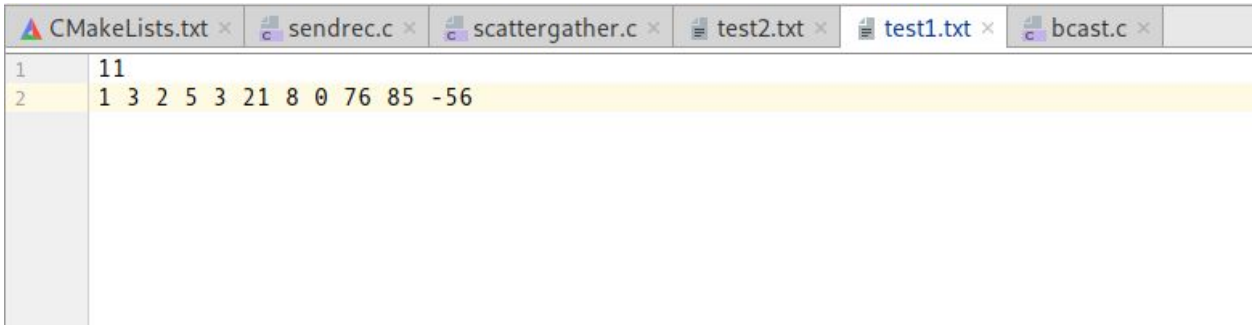
The main **project idea** was mainly suggested by Yasmine. We had a discussion and agreed upon it. The **idea of the implementation** was always discussed among the four of us.
In terms of **code implementation** for the milestones, we always took turns in typing portions of the code. The **reports** were written and discussed by the four of us. The **screenshots** were captured by Menna.

3) User Guide

First you have to choose which test file to run
Below is the code snippet for reading the test file containing the input. The file name can be adjusted in the third line (first parameter) to run different test cases.

```
FILE *pFile;
int n;
pFile = fopen("test1.txt", "r");
fscanf(pFile, "%d", &n);
if (rank == 0) {
    for (int i = 0; i < n; i++)
        fscanf(pFile, "%d", &a[i]);
}
```

Example test case



Then, you have to open the terminal and execute cmake, make and mpirun -n number of processors ./name of executable.
Here are example screenshots

```
[csen@lab64pc5 Microprocessors-master]$ cmake .
-- The C compiler identification is GNU 7.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Found MPI_C: /usr/lib/openmpi/libmpi.so (found version "3.1")
-- Found MPI: TRUE (found version "3.1")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/csen/CLionProjects/Microprocessors-master
```

```
[csen@lab64pc5 Microprocessors-master]$ make
Scanning dependencies of target sg
[ 12%] Building C object CMakeFiles/sg.dir/scattergather.c.o
[ 25%] Linking C executable sg
[ 25%] Built target sg
Scanning dependencies of target sendrec
[ 37%] Building C object CMakeFiles/sendrec.dir/sendrec.c.o
[ 50%] Linking C executable sendrec
[ 50%] Built target sendrec
Scanning dependencies of target bcast
[ 62%] Building C object CMakeFiles/bcast.dir/bcast.c.o
[ 75%] Linking C executable bcast
[ 75%] Built target bcast
Scanning dependencies of target serial
[ 87%] Building C object CMakeFiles/serial.dir/serial.c.o
[100%] Linking C executable serial
[100%] Built target serial
```

```
[csen@lab64pc5 Microprocessors-master]$ mpirun -n 4 ./sg
-56 0 1 2 3 3 5 8 21 76 85 [csen@lab64pc5 Microprocessors-master]$
```

And then you can see the output of the program (a sorted array of integers)

4) Discussion of the obtained results:
The results suggest that the sequential is the fastest approach followed by scatter and gather followed by MPI send and receive then finally broadcast was the slowest.

Regarding the memory, Scatter and gather significantly uses less memory since the array is distributed evenly over all processes in contrast to the other methods where the whole array is sent to each processor

A possible explanation to the performance is:

- Broadcast: it seems the broadcast operation is too slow

- MPI send and receive: it would be efficient only if there is a huge array size with a lot of processors, otherwise, the overhead that comes with parallel programming is greater than the time the parallel programming saves

- Scatter and Gather: it is memory efficient and thus it is the fastest of the parallel programming methods, however, the same problem with the overheads related to the parallel programming paradigm still persists

And the following screenshots illustrate what this explanation numerically

Time analysis for **sequential** code:(memory: ~2n)

```
real    0m0.089s
user    0m0.041s
sys     0m0.003s
```

Time analysis for the Parallelized program using **send and receive** and 4 cores:(memory: ~ no. of processes * n)

```
real    0m2.966s
user    0m11.237s
sys     0m0.409s
```

Time analysis for the Parallelized program using **Broadcasts** and 4 cores:(memory: ~ no. of processes * n)

```
real    0m15.021s
user    0m58.265s
sys     0m1.600s
```

Time analysis for **Scatter and Gather**: (memory: ~3n)

```
real    0m0.298s
user    0m0.828s
sys     0m0.135s
```