COURSE: CMPE300
NAME: Hamza AKYILDIZ
PROJECT TITLE: MPI Project
PROJECT TYPE: Programming Project
DATE: 20.01.2021

In this project, we are responsible for implementing parallel programming with C/C++ using MPI library. We will implement a parallel algorithm for feature selection using *Relief Algorithm*.

The program is executed by commands that are special for MPI environment. To compile the code with MPI environment following command:

```
mpic++ -o main ./main.cpp
```

should be executed to create an object file with the given name *main* with the code in the given address *./main.cpp*. After object file is created, following command:

```
mpirun --oversubscribe -np N main
```

runs the program with *N* processes which consist of a master process and *N-1* slave processes. However, program does not start in N distinct processes. Instead, master process runs the code, and forks the slave processes in progress. Program terminates itself without any user interaction after execution.

The program reads a *.tsv (tab-separated values)* file. The input will be as follows:

```
P
N       A       M       T
... N lines of input data
```

P is the total number of processors (1 master, P-1 slave). N is the number of instances. A is the number of features. For each instance, after A features, there is a single class variable which is 0 or 1 in this project. We will have 0 or 1 as target class value. M is the iteration count for weight updates. T is the number of features selected. The rest is N lines of input data. Slave processes give the indexes of features that was chosen by *Relief Algorithm* as an output. Master processes gathers the indexes of the features that was determined in slave processes and gives a result as an output. Outputs are written on the terminal.

The program starts executing by declaring variables as follows:

```
int rank, numprocs;
int N,A,M,T;
int dataPart;
double** dataSet;
int* classTag;
```

COURSE: CMPE300
NAME: Hamza AKYILDIZ
PROJECT TITLE: MPI Project
PROJECT TYPE: Programming Project
DATE: 20.01.2021

After declaring the variables, program sets the MPI environment by calling *MPI_Init(&argc ,&argv), MPI_Comm_size(MPI_COMM_WORLD, &numprocs), and MPI_Comm_rank(MPI_COMM_WORLD, &rank)*. After this point, program divide into two separate parts for master and slaves.

In master processes, the program reads input file's first two lines into the variables *N, A, M, T*. Initialize the *dataPart* which is the number of lines for which each process is responsible. The rest *N*A* of the input file is read into *dataSet* which is allocated dynamically as 2D array with a size of *N* and *classTag*. Master process broadcast the values *dataPart, A, T, M* to the slave process. Slave processes also call *MPI_Bcast* and puts the values that was broadcasted into addresses in their separated memory spaces. Master process sends the *dataSet* partitions with size of *A*dataPart* and *classTag* partitions with size of *dataPart* to the slaves by using *MPI_Send.* Master process collects the data by calling *MPI_Recv* with size of *T* and puts them into a set. Finally, it writes the selected features into the screen.

In slaves' processes, they call *MPI_Bcast* to receive the values that was broadcasted by master process. They dynamically allocate 2D array with the size of *dataPart*A* and an array with size of *dataPart.* After initialization of memory spaces to process the data in the program, slave processes receive the data by calling *MPI_Recv* from master process and puts them in to the allocated space of their memory spaces. Executing following lines initializes the variables used in the slave process with the given sizes.

```
dataSet=dynamic2DArray(dataPart, A)
classTag=new int[dataPart]
double* W = new double[A]
double* minA = new double[A]
double* maxA=new double[A]
```

maxA and *minA* arrays keep the maximum and minimum of the weight of the features in the given index in all instances. At the beginning of the iteration where I select target hit which is in *row1*

COURSE: CMPE300
NAME: Hamza AKYILDIZ
PROJECT TITLE: MPI Project
PROJECT TYPE: Programming Project
DATE: 20.01.2021

and for the other instances that are used for finding nearest hit and miss is in *row2*. Following lines are executed in every iteration since in every iteration target instance changes.

```
double* row1=dataSet[targetInstance]
double* row2
double nearestHit=1000000,nearestMiss=1000000,distance;
int hitIndex=0.missIndex=0
```

For the first iteration that runs on every index on the instances, program fills the *minA and maxA.* Since *minA* and *maxA* has memory they are updated every time *row2* changes. After calculating distances between the *row1 and row2* by calling *manhattanDistance(row1,row2,A)* program

```
W[A] = W[A] - diff(A, targetInstance , nearistHit)/M + diff(A, targetInstance, nearistMiss)/M
```

checks if it is a miss or hit and updates the *nearestHit* or *nearestMiss* if the distance is shorter than the kept value. After every iteration, the *W* will be updated according to following formula:

And the *diff* is calculated according to following formula:

$$\text{diff(A,I}_1, I_2) = \frac{|value(I_1,A) - value(I_2,A)|}{max(A) - min(A)}$$

After *M* iteration, slaves calculated the weights of the features according to their data partitions. Finally, slaves determine *T* biggest weight in their *W* arrays and their indexes and put them into the following arrays. Before sending the *outputIndex* to the master process, slave processes sort

```
double* output = new double[T]
int* outputIndex = new int[T]
```

them in ascending order for the sake of the printing order. At the end of the slave part, slaves send their selected features to the master process by using *MPI_Send.* Since receive and send are blocking functions when master call *MPI_Recv* it waits until it gets the results from the slave. Therefore, in any case master needs to wait slaves before termination.

MPI_Init(&argc,&argv) which is passed the addresses of the command line arguments number and the command line arguments. *MPI_Init* forks the child processes and after this point program becomes a parallel program that is separately executed by *P* processes. We have default communicator item that is available in <*mpi.h*> and called *MPI_COMM_WORLD*.

The call to *MPI_Comm_size(MPI_COMM_WORLD, address)* puts the number of the processes in the communication into the given address.

COURSE: CMPE300
NAME: Hamza AKYILDIZ
PROJECT TITLE: MPI Project
PROJECT TYPE: Programming Project
DATE: 20.01.2021

The call to *MPI_Comm_rank(MPI_COMM_WORLD, address)* is to put id of process into the given address in the separate processes. Master process get 0 in the rank.

The call to *MPI_Bcast(address, count, type, source, MPI_COMM_WORLD)* is to broadcast a variable to other processes. First argument is the argument that will be broadcasted or that will keep the value that was broadcasted. Therefore, all processes, either receiver or sender, should call *MPI_Bcast.* Second argument is the count which is simply the number of arguments that will be broadcasted. Third argument is type of the variable that is broadcasted. Fourth argument is the source rank of the broadcasted value. Last one is the communicator.

The call to *MPI_Send(address, cout, type, dest, tag, MPI_COMM_WORLD)* is to send the message to the specific process by using it rank which is declared in the fourth argument of the function as *dest.* First three argument is the same as with MPI_Bcast. *tag* can be considered as message id in case there is more than one message to send the same process. This is a blocking send.

The call to *MPI_Recv(address, cout, type, source, tag, MPI_COMM_WORLD, status)* is to receive a message from a specific source. This has *source* instead of *dest* and also has a last argument which is *status* that keeps the status of the receiving message. This is a blocking receive.

manhattanDistance(double row1, double* row2, int n)* calculates the distance between two instances and returns the distance according to following formula.

$$\text{ManhattanDistance}([x_1, y_1, z_1], [x_2, y_2, z_2]) = |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$$

min_max(double row, double* minA, double* maxA, int n)* calculates the min and max element of the array and puts them into *minA* and *maxA* in the relative index of the .

dynamic2DArray(int rows, int cols) allocates 2D array with size of *rows*cols* and returns the allocated array. It allocates 1D array with size *rows*cols* and puts the address of every other *rows* address into the 2D array.

*void sort(int *arr, int size)* sorts the array in ascending order by using selection sort.

COURSE: CMPE300
NAME: Hamza AKYILDIZ
PROJECT TITLE: MPI Project
PROJECT TYPE: Programming Project
DATE: 20.01.2021

In the project we can use more efficient sorting algorithm since selection sort always work with quadratic time complexity. For finding the *T* best weights we can use heap which decrease time complexity from *T*A* to *T*logA.* The sake of the requirement that master process will outputs at the end, using *MPI_Barrier* would be unnecessary since master calls *MPI_Recv* for every slave which blocks the master. Allocating dynamic 2D array is a challenge. Without allocating it, MPI_Send and MPI_Recv functions should have been called for every instance, which would be pretty time consuming.

The main purpose of the program is to add a parallel perspective to the relief algorithm used for feature selection. I used dynamic 2D arrays for faster communication, which reduces the number of times of the call to communication functions, *MPI_Send* and *MPI_Recv.*

I used Mac OS 11.1 environment with Apple clang version 11.0.0 and open MPI 4.0.3. To compile the following input lines, we should first run the following command on terminal, otherwise it may give an system call error;

```
export OMPI_MCA_btl=self,tcp
```

Compiling the code;

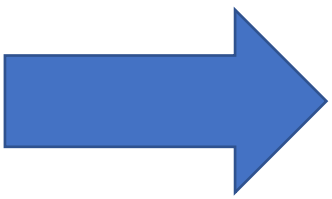
```
mpic++ -o cmpe300_mpi_2019400249 ./cmpe300_mpi_2019400249.cpp
```

Running the executable with;

```
mpirun --oversubscribe -np 3 ./cmpe300_mpi_2019400249 <inputFile>
```

```
3
10 4 2 2
6.0 7.0 0.0 7.0 0
16.57 0.83 19.90 13.53 1
0.0 0.0 9.0 5.0 0
11.07 0.44 18.24 15.52 1
5.0 5.0 5.0 7.0 0
16.55 0.25 10.68 17.12 1
7.0 0.0 1.0 8.0 0
17.44 0.01 18.55 17.52 1
5.0 3.0 4.0 5.0 0
16.80 0.72 10.55 13.62 1
```

Example input



```
Slave P1 : 2 3
Slave P2 : 0 2
Master P0 : 0 2 3
```

Example output