CMPE 300

Analysis of Algorithms

MPI Programming Project

Ece Dilara Aslan

Submission Date: 27.01.2021

a. Introduction

In this project, the main idea is selecting the most related features of the given class with Relief algorithm. It is done via parallel programming. First of all, master processor reads input data from the file and scatters them to the slave processors. Slave processors calculate weights of each feature according to their data. There are three helper functions which are *findMaxMin*, *manhattanDistance* and *findHitMiss*. They support the Relief algorithm in slave processors. After calculating the weights of the features, each processor chooses top weighted features and sends them to the master processor. Master processor gathers them and eliminates the repetition. Final feature set is the most related features of the given class with respect to given data set.

b. Program Execution

While executing the program, there are a couple of necessary inputs. There must be a file consists of number of processors, number of instances, number of features, number of iterations in Relief algorithm, number of top weighted features per slave processor and finally feature values and class labels of instances. The name of this file must be sent via command line. Program outputs ids of top weighted features found in Relief algorithm in slave processors to the console.

In this program, version 4.0.5 of Open MPI and version 11.0.3 of Apple clang (clang-1103.0.32.62) are used.

An example of the program with the given input file mpi_project_dev2.tsv:

The command to compile is;

mpic++ -o mpi mpi.cpp

to run is;

mpirun --oversubscribe -np 11 mpi mpi_project_dev2.tsv

the output is;

```
Ece-MacBook-Pro:CMPE300-MPI-Project-Test-Files ecedilaraaslan$ mpic++ -o mpi mpi.cpp
Ece-MacBook-Pro:CMPE300-MPI-Project-Test-Files ecedilaraaslan$ mpirun --oversubscribe -np 11 mpi mpi_project_dev2.tsv
Slave P2 : 0 3 4 8 11 13 21 26 32 39
Slave P3 : 0 3 4 5 11 18 21 26 46 47
Slave P5 : 0 3 5 11 16 18 21 26 35 47
Slave P6 : 0 3 5 8 16 21 26 30 35 47
Slave P7 : 0 2 3 4 5 16 18 21 32 45
Slave P9 : 0 3 5 11 18 21 26 30 35 47
Slave P10 : 0 5 8 11 16 18 20 21 30 46
Slave P1 : 4 5 8 11 18 21 30 32 44 49
Slave P4 : 0 3 11 14 21 28 30 32 39 40
Slave P8 : 0 3 11 18 21 24 26 39 44 46
Master P0 : 0 2 3 4 5 8 11 13 14 16 18 20 21 24 26 28 30 32 35 39 40 44 45 46 47 49
Ece-MacBook-Pro:CMPE300-MPI-Project-Test-Files ecedilaraaslan$
```

c. Program Structure

First of all, the program includes **mpi.h**, **iostream**, **fstream**, **cmath**, **algorithm**, **thread** and **chrono**. **mpi.h** consists of functions associated with MPI envrionment. **iostream** provides printing to the console, **fstream** provides reading from a file, **cmath** provides *abs*

function, **algorithm** provides *copy*, *fill*, *sort*, *max_element* functions, **thread** provides *sleep_for* function and **chrono** provides *seconds*.

There are six global variables which are named as **p** (number of processors), **n** (number of instances), **a** (number of features), **m** (number of iterations in Relief algorithm), **t** (number of top features per slave processor) and **instance** (number of instances per slave processor). They are declared as global because they may be needed in the helper functions.

In main function, MPI environment and **rank** (rank of the processor) are initialized. Master processor has the rank 0, slave processors have 1 to p-1. Each processor reads first three numbers from the input file and initializes **p**, **n** and **a**, and also calculates and initializes **instance**. If current processor is a slave processor, then work of the file stream is done because other input data will be sent from master processor using message passing.

Four one-dimensional arrays -**features**, **subfeatures**, **topFeatures**, **subTopFeatures**- are declared. **features** will store all of the given instance data and an unused space at the beginning, **subfeatures** will store a subset of the given instance data associated with the current processor, **topFeatures** will store ids of all top weighted features founded by the slave processors and an unused space at the beginning, **subTopFeatures** will store ids of the top weighted features founded by the current processor. **features** and **topFeatures** include an unused space at the beginning. Because when using *MPI_Scatter* or *MPI_Gather* functions, data is scattered/gathered to/from all of the processors. However, in this program it is aimed to slave processors deal with the data and master processor does not process data. So, in order to scatter/gather the right part of data to/from only slave processors, **features** and **topFeatures** have an unused space at the beginning.

In master processor, rest of the input file is read and **m**, **t** and **features** are initialized. **features** stores feature values and class labels of each instance in a sequential way. After the unused space, first **a** indices store feature values of first instance, **a**+1st index has the class label of first instance, next **a** indices store feature values of second instance, next index has the class label of the second instance… File is completely read, so work of the file stream is done. Master broadcasts values of **m** and **t**, and also scatters **features** to each slave's **subfeatures**.

In slave processors, the main work is done, calculating weights of each feature using Relief algorithm. Three one-dimensional arrays -**weights**, **max**, **min**- are declared. **weights** will store the weights of the features in the current processor, **max** will store maximum value of each feature considering instances in the current processor, **min** will store minimum value of each feature considering instances in the current processor. **weights** is initialized with zeros. **max** and **min** are initialized in the *findMaxMin* helper function. This function takes three parameters, feature values of instances in the current processor and two arrays that will be initialized. First, function copies the feature values of first instance into **max** and **min**. Then, for each instance it checks whether the value of a feature of that instance is larger than current maximum value or smaller than current minimum value. At the end of the function, **max** and **min** are initialized with the proper values.

Then, weight calculation iterations start. In each iteration, there is a target instance. This instance is stored in an array called **targetInstance**. After initializing **targetInstance**, nearest hit and nearest miss of this target must be found. Nearest hit and miss are stored in arrays called **hit** and **miss**, respectively. These arrays are initialized in *findHitMiss* helper

function. This function takes five parameters, feature values of instances in the current processor, feature values of target instance, order of the target instance, two arrays that will be initialized. In *findHitMiss* function, there are two local variables which are called **hitDist** and **missDist**. These variables store distance of current nearest hit and distance of current nearest miss with respect to target instance. There is a loop that iterates through instances, **temp** stores feature values and class label of current instance. If current instance is not the target instance, then distance between this current instance and target instance is calculated via *manhattanDistance* helper function. *manhattanDistance* calculates Manhattan distance between its parameters and returns. **distTemp** stores returned value from this function. If the current instance is a hit, then it is checked whether the current one is the nearest. Likewise, if the current instance is a miss, then it is checked whether the current one is the nearest. At the end of the *findHitMiss* function, **hit** and **miss** are initialized with the feature values of proper instances. After finding nearest hit and miss, for each feature diff of **targetInstance** and **hit** is stored in **diffHit** and diff of **targetInstance** and **miss** is stored in **diffMiss**. Then, weight of the current feature is updated according to the given calculations.

After reaching the final weights of features, ids of largest **t** of them must be printed. **ptr** is a pointer to the beginning of the **weights** array. For **t** times, maximum value in the **weights** array is founded and its order -at the same time its id- is stored in **subTopFeatures**. In each iteration, maximum value is changed with the smallest possible value of long double type because the next maximum value must be found in each iteration. After finding top weighted features, their ids are sorted in ascending order. There is a *MPI_Barrier* function between finding and writing top weighted features because each processor should complete its writing at a time, so all of the allocated time for a processor is used for writing.

Master gathers top weighted features from slaves, gathers **subTopFeatures** from all slave processors and stores in **topFeatures**. Then, master processor sleeps for 1 second in order to guarantee that writes after slave processors. After sleeping, it sorts the ids in **topFeatures** in ascending order. **tempFeature** stores the id of current feature that is written to the console. For each feature id in **topFeatures**, if it is different from **tempFeature**, then it is written and **tempFeature** is updated. By doing that, repetition of ids is avoided.

At the end of the code, there is a *MPI_Barrier* function which provides synchronization before finishing the MPI environment. MPI environment and whole program is finished.

d. Difficulties Encountered

I spend a lot of time trying to use a two-dimensional array while storing feature values and class labels of all instances. I thought that two-dimensional array is a good choice because feature values and class labels can be stored instance by instance. However, utilizing a two-dimensional array is difficult like sending it as a parameter or scatter it to slave processors. So, I have decided that these data can be stored in an one-dimensional array and by making some effort while accessing an index of it, it can be used easily. Also, variable type is an important point of this project because when one accidently declare a parameter that should have a floating point type as an integer, outputs behave similar but not exactly the same with the correct output. I have spent some time while trying to find my mistake and it was because of a type declaration of a parameter.

e. Conclusion

To sum up, the aim of this project is finding the most related features of a class making use of the data that is given. While finding out which features are most related, parallel programming is used and by doing so workload is divided into different processors. Each slave processor finds and writes top weighted features according to its dataset. At last, master processor gathers these top weighted features from all slave processors and writes all of them excluding repetitions. Therefore, for each partition of data and for the whole data top weighted features are founded.