20.01.2021

CMPE300 ANALYSIS OF ALGORITHMS

MPI Parallel Programming Project

Muhammed Göktepe 2017400162

To: Nuriye Özlem Özcan

Introduction

This project is about a parallel algorithm for future selection using Relief. In my solution I used C++ with MPI library. At the beginning, input file is read, and necessary parameters are assigned. Later, instances are partitioned among processes and implemented Relief algorithm as number of iteration times. Lastly, selected futures are sorted and printed.

Program Interface

To execute the program, you must be on Linux environment. You need to have a C++ compiler. If you do not have any you can install it via terminal. Then, you should install and setup the openmpi to your computer. In Linux you can install it via terminal with "***sudo apt install openmpi-bin***" command. Before starting the program, you must change the working directory in the terminal to the folder where the executable program is. Then you can start the program with one parameter which is the absolute path of input file. First you should compile the program with this command. "***mpic++ -o outputpp ./cmpe300_mpi_2017400162.cpp***" Following command will start the program. "***mpirun --oversubscribe -np <number of processes> outputpp <input file's absolute path>".*** If program stalls you can exit the program in terminal using Ctrl + C shortcut.

Program Execution

This program takes one input file which contains numbers only. In this file there are plenty of instances and their features. This program gives you the most relevant features that you can use in decision models. As an output this program prints some number of feature id's that are most contribute to your prediction variable.

Input and Output

This program takes one file as an input and prints the output to the console. Input file consisting of multiple lines of numbers. In the first line there is one integer value which is the number of processors in the program. In the second line there are four numbers which are the number of instances (N), the number of features (A), the iteration count for weight updates (M) and resulting number of features (T), respectively. And there are N lines in the input file after these two lines. These N lines represent N instance, and each instance has A features. After A features, there is one class variable which is 0 or 1. For example.

```
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
```

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

This is an example of an input file format. If we look at the output format it will consist of P lines. In each line there are T numbers which represents most relevant features. As an example.

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

Master processor combines slave processors outputs and prints them.

Running and compiling example:

mpic++ -o outputpp ./cmpe300_mpi_2017400162.cpp

mpirun --oversubscribe -np 6 outputpp /home/gktpmuhammed/Desktop/CLionProjects/CMPE300_Project1/ mpi_project_dev0.tsv

```
gktpmuhammed@gktpmuhammed-VirtualBox:~/Desktop/CLionProjects/CMPE300_Project1$ mpirun --oversubscribe -np 6 outputpp ./mpi_project_dev0.tsv
Slave P4 : 5 7
Slave P2 : 0 2
Slave P1 : 0 5
Slave P3 : 5 7
Slave P5 : 0 3
Master P0 : 0 2 3 5 7gktpmuhammed@gktpmuhammed-VirtualBox:~/Desktop/CLionProjects/CMPE300_Project1$
```

mpic++ -o outputpp ./cmpe300_mpi_2017400162.cpp

mpirun --oversubscribe -np 6 outputpp /home/gktpmuhammed/Desktop/CLionProjects/CMPE300_Project1/ mpi_project_dev1.tsv

```
gktpmuhammed@gktpmuhammed-VirtualBox:~/Desktop/CLionProjects/CMPE300_Project1$ mpirun --oversubscribe -np 6 outputpp ./mpi_project_dev1.tsv
Slave P2 : 3 5 7 8 18
Slave P4 : 0 3 4 5 16
Slave P3 : 0 3 12 13 16
Slave P1 : 4 5 8 10 18
Slave P5 : 0 5 6 11 18
Master P0 : 0 3 4 5 6 7 8 10 11 12 13 16 18gktpmuhammed@gktpmuhammed-VirtualBox:~/Desktop/CLionProjects/CMPE300_Project1$
```

mpic++ -o outputpp ./cmpe300_mpi_2017400162.cpp

mpirun --oversubscribe -np 11 outputpp /home/gktpmuhammed/Desktop/CLionProjects/CMPE300_Project1/ mpi_project_dev2.tsv

```
gktpmuhammed@gktpmuhammed-VirtualBox:~/Desktop/CLionProjects/CMPE300_Project1$ mpirun --oversubscribe -np 11 outputpp ./mpi_project_dev2.tsv
Slave P1 : 4 5 8 11 18 21 30 32 44 49
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 P8 : 0 3 11 18 21 24 26 39 44 46
Slave P10 : 0 5 8 11 16 18 20 21 30 46
Slave P9 : 0 3 5 11 18 21 26 30 35 47
Slave P5 : 0 3 5 11 16 18 21 26 35 47
Slave P7 : 0 2 3 4 5 16 18 21 32 45
Slave P6 : 0 3 5 8 16 21 26 30 35 47
Slave P4 : 0 3 11 14 21 28 30 32 39 40
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 49gktpmuhammed@gktpmuhammed-VirtualBox:~/Desktop/CLionProjects/CMPE300_Proj
ect1$
```

Program Structure

There is one cpp file and all necessary work done in there. There is a main method and one comparator. In main program starts with initializing MPI environment.it gets the number of processors and rank of the processors. Then it takes the first argument which is the absolute path of the input file from arguments. It starts to parse the input file. First it takes the number of processors and then take number of instances, number of features, number of iterations and resulting number of features from first two lines of input file. Program broadcasts these five values to the all the processors. Then it creates two arrays. One is for storing whole instances in the input file. And the other is storing partitioned instances to the processors. Then if the processor is root processor than it continues to read file and stores all the feature and class values in the *array*. After reading the file it closes the file stream. All instances partitioned among processors and send to them by *MPI_Scatter* function. There is a while loop. Relief algorithm implemented in this loop. It runs until it receives root function is reached. At first, there are two array initializations. *master_sorted_array* stores the features that have higher weights calculated in each processor. *sorted_array* used in slave processors. It stores features that have higher weights. All slaves *sorted_array's combines and creates master_sorted_array.* If the processor is a slave processor first if statement implements Relief algorithm to the instances. First it creates *weight_array* with initial value of zero. *max_array* stores the max value for each feature among all instances of current processor. *min_array* stores the min value for each feature among all instances of current processor. *sorted_weights* vector stores selected features after implementing Relief algorithm. First for loop will run by number of iterations times. It creates *one_line_array* and stores single instance in it with the second for loop. *hit_line* stores nearest hit feature's id. *miss_line* stores nearest miss feature's id. *hit_sum* stores minimum Manhattan distance between two instances in same class (hit). *miss_sum* stores minimum Manhattan distance between two instances in different classes (miss). Third for loop iterates among all instances of current processor and calculates Manhattan distance between target instance and all instances. First if statement in this loop prevents program comparing the instance with itself. In fourth for loop it implements Manhattan distance between two instance and updates *min_array* and *max_array* for each feature. Then if calculated Manhattan distance is smaller than previous ones it updates *hit_sum, hit_line, miss_sum, miss_line* based on current compared instances class variables. Then it updates *weight_array* based on *hit_line* and *miss_line* instances. From now on it is all about printing and sorting features. *MPI_Gather* combines all *sorted_arrays* and places them to *master_sorted_array.* Now root processor starts to execute and puts the values in the *master_sorted_array* to set to eliminate duplicates. It is started from *resulting_num_of_features* because *MPI_Gather* take values from all processors including root processor but we make calculations in slave processors. So, we ignore first *resulting_num_of_features* in *master_sorted_array.* Then we copy the values to array for sorting them in increasing order. And finally, it prints the values and finalize MPI environment.

Examples

2
4 3 2 2
9 2 2 0
5 1 0 0
9 3 2 1
9 3 3 1

There are 2 processors 1 master 1 slave. There are 4 instances, and each instance has 3 features. In this program there will be 2 iterations and program want from us to find two most relevant features. First of all program sends four instances to slave P1. In first iteration, processor selects first instance (9 2 2 0) and compare it with others and finds nearest hit and miss using Manhattan distance. (|9-5| + |2-1| + |2-0| = 7) Our hit is the second instance (5 1 0 0) and nearest miss is third instance (9 3 2 1).

Now it updates *weight_array.* Current weights are zero.

0^{th} feature = $0 - \left(\frac{\frac{|9-5|}{9-5}}{2}\right) + \left(\frac{\frac{|9-9|}{9-5}}{2}\right) = -0.5$

1^{st} feature = $0 - \left(\frac{\frac{|2-1|}{3-1}}{2}\right) + \left(\frac{\frac{|2-3|}{3-1}}{2}\right) = 0$

2^{nd} feature = $0 - \left(\frac{\frac{|2-0|}{3-0}}{2}\right) + \left(\frac{\frac{|2-2|}{3-0}}{2}\right) = -0.33$

In second iteration processor selects 2^{nd} instance (5 1 0 0) and makes same calculations like first one. Nearest hit is 1^{st} instance (9 2 2 0) and nearest miss is 3^{rd} instance (9 3 2 1).

Now it updates *weight_array*. Current weights are -0.5, 0, -0.33.

0^{th} feature = $-0.5 - \left(\frac{\frac{|5-9|}{9-5}}{2}\right) + \left(\frac{\frac{|5-9|}{9-5}}{2}\right) = -0.5$

1^{st} feature = $0 - \left(\frac{\frac{|1-2|}{3-1}}{2}\right) + \left(\frac{\frac{|1-3|}{3-1}}{2}\right) = 0.25$

2^{nd} feature = $-0.33 - \left(\frac{\frac{|0-2|}{3-0}}{2}\right) + \left(\frac{\frac{|0-2|}{3-0}}{2}\right) = -0.33$

So we make two iterations now we need to selected top two features and print them. Most relevant feature is first one with 0.25 weight and second one is 2^{nd} feature with -0.33 weight. So slave processor prints

Slave P1 : 1 2

And master process also prints

Master P0 : 1 2

Improvements and Extensions

At the beginning I planned to use 2d array instead of 1d array but I can't use MPI_Scatter and Mpı_Gather functions with 2d array. So, that can be improved. Weak point is if there can be a division by zero program crushes.

Difficulties Encountered

Debugging is complicating end exhausting in parallel programming. Also, it is a new concept and new library so learning their features can take some time.

Conclusion

In conclusion using parallel programming we can reduce the run time of a program if programs structure is appropriate for parallel programming. So, the efficiency will increase, and cost will decrease. Using Relief programming we can select most relevant parameters for our study and make more precise decisions in our program.