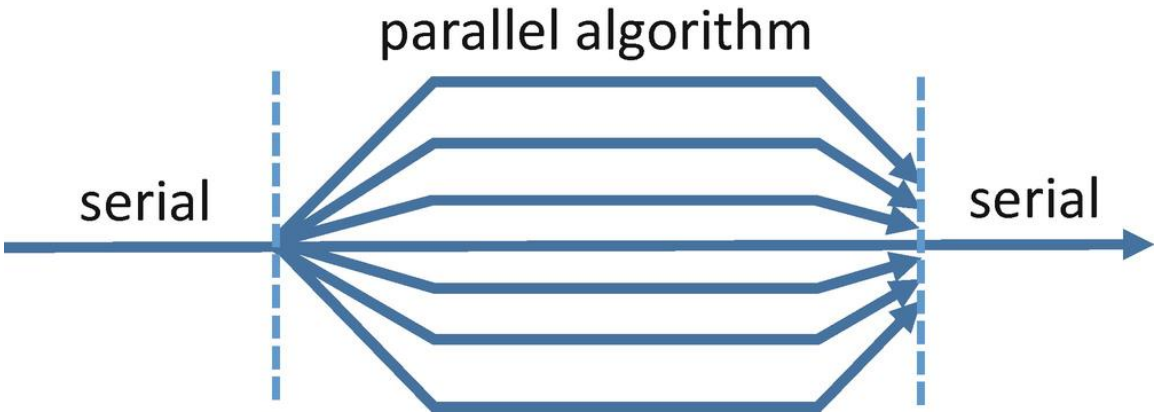

MPI Programming Project

CMPE300 2020-FALL

Orhun Görkem

Introduction

In this Project, our aim was to implement a relief algorithm with the performance increasing effect of MPI (Message Passing Interface). Relief algorithm works for eliminating the unnecessary features for a given class. The program execution is conducted by of one master and several slave processors. Since there can be high number of instances to examine for feature selection, the job of examination should be shared among some additional processors. Since single instruction is executed on multiple data, we can say that SIMD machine is used for this program.

Program Execution

The code is written with Open-MPI 4.0.5. Therefore, installation of Open-MPI should be made before execution.

For really big inputs, to avoid stack overflow, the following command is recommended:

- ulimit -S -s 131072

Then, directory with the cmpe300_mpi_20174001171.cpp file and test cases including the instances should be navigated. After that, the commands below will be sufficient:

- mpic++ -o cmpe300_mpi_20174001171 ./cmpe300_mpi_20174001171.cpp
- mpirun --oversubscribe -np <P> cmpe300_mpi_20174001171 <inputfile>

where P is number of processors to use and input file is the relative path for input file.

Input file should include:

- Number of processors to use : P
- Number of instances to apply relief : N
- Number of features in each instance: A
- Number of iterations in relief algorithm : M
- Number of features to select for each processor : T
- N lines of input that keeps all features and classes of each instance

To visualize, input looks like this:

P

N A M T

<Inst 1>

<Inst 2>

<Inst 3>

After execution, program outputs the selected feature id's for each processor and a line of master processor output which unions all the yielded features ids by slaves.

Example for compiling and execution:

```
orhun@orhun-VirtualBox:~/cmpe300/MPI-PS/Project$ ulimit -S -s 131072
orhun@orhun-VirtualBox:~/cmpe300/MPI-PS/Project$ mpic++ -o cmpe300_mpi_2017400171 ./cmpe300_mpi_2017400171.cpp
orhun@orhun-VirtualBox:~/cmpe300/MPI-PS/Project$ mpirun --oversubscribe -np 11 cmpe300_mpi_2017400171 mpi_project_dev2.tsv
Slave P6 : 0 3 5 8 16 21 26 30 35 47
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 P4 : 0 3 11 14 21 28 30 32 39 40
Slave P8 : 0 3 11 18 21 24 26 39 44 46
Slave P3 : 0 3 4 5 11 18 21 26 46 47
Slave P1 : 4 5 8 11 18 21 30 32 44 49
Slave P5 : 0 3 5 11 16 18 21 26 35 47
Slave P2 : 0 3 4 8 11 13 21 26 32 39
Slave P7 : 0 2 3 4 5 16 18 21 32 45
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 49orhun@orhun-VirtualBox:~/cmpe300/MPI-PS/Project$
```

Program Structure

The program starts with sequential input reading by the master processor. Then the relief iterations are conducted by partitioning the instances to slave processors in parallel manner. At the end, master gathers the results and outputs the cumulative answer.

Part 1 (Sequential): The necessary inputs like P, N, A, M, T are read from the file. Number of instances per processor (instPerProcessor) is calculated. (equal number of instances for each slave) The necessary data structures are initiated with relevant sizes.

- float **instGetter**[instPerProcessor][A]: This 2D float array keeps the instance features in each processor. Therefore, it is the receiver buffer for MPI_Scatter.
- float **classGetter**[instPerProcessor]: This 1D float array keeps the classes of instances in each processor. Therefore, it is the receiver buffer for the second MPI_Scatter call.
- float **attributes**[N+instPerProcessor][A]: This 2D float array keeps all the input instances read from file. This is the sender buffer for the MPI_Scatter. First instPerProcessor indices are scattered to master, but we do not want master to receive instances to apply relief, this job only belongs to slaves. That is why, first instPerProcessor indices are kept as null while input is being read. In short, this array keeps null indices at first indices and instance features at remaining indices.
- int **classes**[N+instPerProcessor]: Similar to attributes but keeps the classes for instances.
- int **featureGather**[A*P]: This will keep the selected features after MPI_Gather. Therefore, it is the receiver buffer for that call. Inştiated at the beginning but will be used at the end.
- int **result**[A]: The array that keeps the selected features in ascending order in each processor. Processors fill their own result array parallelly and the content is gathered in the featureGather at the end. Hence, result is the sender buffer of MPI_Gather.

Then the input of instances is read into attributes and classes with nested for loop. Since the data is ready now, these calls are executed:

- **MPI_Scatter(attributes , instPerProcessor*A , MPI_FLOAT , instGetter, instPerProcessor*A,MPI_FLOAT, 0 , MPI_COMM_WORLD);**
- **MPI_Scatter(classes , instPerProcessor , MPI_INT , classGetter , instPerProcessor , MPI_INT , 0 , MPI_COMM_WORLD);**
- **MPI_Bcast(&M , 1 , MPI_INT , 0 , MPI_COMM_WORLD);**
- **MPI_Bcast(&T , 1 , MPI_INT , 0, MPI_COMM_WORLD);**

MPI_Scatter processes the send and receive buffers with row-major order. Therefore when we use attributes 2d array as sender, instGetter as receiver and instPerProcessor*A as data count, attributes are partitioned into 1d array of floates with size instPerProcessor*A and those partitions are written onto instPerProcessor indices of instGetter. Since each index in instGetter has size A, all partitioned float values fall into correct places.

MPI_Bcast is used to broadcast M and T values to slaves.

Part 2 (Parallel): In this part, slaves work parallel to execute relief iterations with different sets of instances. Each slave fetch the scattered data with instGetter. Weights array is initiated with zeros. Then the relief iterations begin. An instance is picked and nearest miss and hit is found in every iteration.

Nearest miss to instance i : The instance with the lowest Manhattan Distance to i among the instances that are not in the same class with i.

Nearest hit to instance i : The instance with the lowest Manhattan Distance to i among the instances that are in the same class with i.

float ManhattanDistance(vector<float> v1, float* v2) : The helper function that calculates manhattan dşstance between given v1 and v2.

After that, weights array is updated with the following formula:

weights[j]=weights[j]-abs(instance[j]-hitInst[j])/maxdif[j]+ abs(instance[j]-missInst[j])/maxdif[j];

where hitInst = nearest hit and missInst = nearest miss (formula is applied with all features j) and

where max is the highest value of the feature among all the instances that are sent to corresponding slave processor and min is the lowest.

Weights are updated with M iterations. Then features with highest weights are picked. To do that, weights and feature ids are paired in a vector and sorted. Top T feature ids are picked, put in result array and sorted in ascending order. Then the output is printed.

Part 3 (Sequential): Results of all slaves are collected into featureGather with following call:

- **MPI_Gather(result, T , MPI_INT , featureGather , T , MPI_INT , 0, MPI_COMM_WORLD);**

featureGather may keep duplicate feature ids, that's why, program stores its content into a set. Set has no duplicates. Then set content is copied into a vector to output the correct result (which is the union of selected features by each processor).

Difficulties Encountered

Getting used to the MPI syntax and procedure was really challenging. Some calls are needed to be made in sequential area while some others belong to parallel areas. For Example, I tried to call MPI_Scatter in master processor part (where rank=0), which appeared to be wrong since MPI_Scatter also scatters some data to the master. On the other hand, implementation of relief algorithm was not intuitive since I had no idea about that before. Therefore, I implemented the procedure with some little mistakes at first.

Conclusion

This project helped me to get used to parallel programming best practices. Also, it was nice to being familiar to MPI framework since I believe this going to be very useful in my future experiences. I would like to return back and see what I have done after learning Machine Learning concepts which will help me to interpret the idea behind the job.