CMPE300.02

ANALYSIS OF ALGORITHMS

PROGRAMMING PROJECT

MAP REDUCE WITH MPI

BERK EROL

2014400189

SUBMITTED TO: TUNGA GÜNGÖR

SUBMISSION DATE: 17/12/2017

INTRODUCTION

In this project, we are designing and building a program that finds how many times does each word occur in the input and sorts the result according to the lexicographical order of words. We use MapReduce algorithm for this project. As you might guess, this process consists of two steps: map and reduce. First, we map words to their counts (all of them are 1) then sort the mappings according to the lexicographical order of their words. Second, we count the occurrences of each same mapping (they are all after one another) then reduce them to unique mappings by adding the counts of the same mappings.

In order to utilize parallel programming, we use a master processor and slave processors that are used by this master processor. Complete execution is as follows:

- Master reads the input from the file. Then it splits the words and sends them to slaves.
- Slaves map words to the counts (all of them are 1) then send these mappings back to the master.
- Master splits the mappings and sends them to slaves.
- Slaves sort mappings according to the lexicographical order of their words then send these sorted mappings back to the master.
- Master gathers all sorted mappings and merges them. Then it reduces mappings with the same words into one mapping by adding up their number of occurrences and it produces the output which consists of sorted unique mappings.

PROGRAM INTERFACE

This program is written with C++ using Open MPI Project. It has two optional arguments: input file name and output file name. If we want to use arguments, we must give both of them. Otherwise, “speech_tokenized.txt” is the default input file name and “reduced.txt” is the default output file name. It can be compiled with the Open MPI’s wrapper compiler “mpic++”. It can be run with “mpirun” with the argument “-n” denoting the total number of processors (master and slaves together). It terminates itself.

- Command to compile:
  - mpic++ -std=c++11 main.cpp
- Command to run:
  - mpirun -n 4 ./a.out input.txt output.txt

PROGRAM EXECUTION

To run the program, user should enter the following command to the terminal which is opened at the same folder.

- mpirun -n <processors> ./a.out <input-file> <output-file>

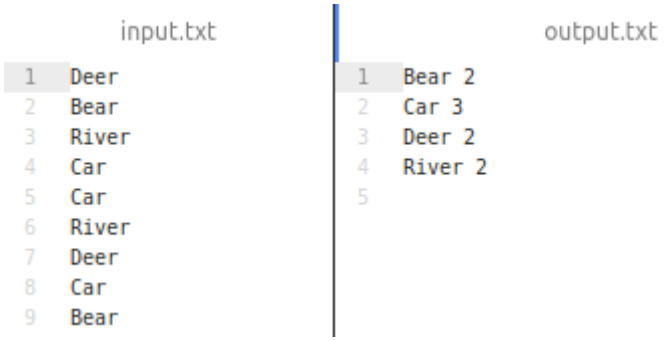
In this command, <processors> is an integer denoting the total number of processors. So there are 1 master and n-1 slave processors. <input-file> and <output-file> are optional arguments that are the names of input and output files respectively. Input file must contain a token in each line. Output file consists of lines which contains token and total occurrence of this token in the input file as an integer.

Briefly, this program finds how many times does each token occur in the input file and produces an output which consists of sorted lines which contains token and its count.

```
berk@BERKEROL:~/MEGAsync/2017-2018 1/CMPE300/application project$ cat input.txt
Deer
Bear
River
Car
Car
River
Deer
Car
Bear
berk@BERKEROL:~/MEGAsync/2017-2018 1/CMPE300/application project$ mpirun -n 4 ./a.out input.txt output.txt
berk@BERKEROL:~/MEGAsync/2017-2018 1/CMPE300/application project$ cat output.txt
Bear 2
Car 3
Deer 2
River 2
berk@BERKEROL:~/MEGAsync/2017-2018 1/CMPE300/application project$
```

INPUT and OUTPUT

The input file must be a text file containing one token in each line. The number of total characters in each token must be less than 50. A token must contain only letters and/or digits. The output file is a text file containing lines which consists of token and its total number of occurrence. These lines are sorted according to the lexicographical order of tokens.



PROGRAM STRUCTURE

DATA STRUCTURES

Word struct:

It consists of a char array of length “MAX_WORD_LENGTH” which is defined as a global constant and is equal to 50. It may seem redundant because it only contains one field but I created it to make the send/receive part easier.

Pair struct:

It consists of a char array of length “MAX_WORD_LENGTH” and an integer denoting the number of occurrences of the word in the input file.

HELPER METHODS

compare_pairs:

It takes two pairs as parameters then returns a boolean (true if first argument is less than second one). It is used for sorting pairs in slaves.

send:

It takes a vector to be sent and an integer denoting the destination processor as parameters. It sends this vector as a byte array. It uses **MPI_Send**(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator) method.

receive:

It takes the type to be received and an integer denoting the source processor as parameters and returns the received vector. First it finds how many bytes to receive from the source then it receives this vector as a byte array (just like how it sends). It uses the following MPI methods:

- **MPI_Probe**(int source, int tag, MPI_Comm communicator, MPI_Status* status)
- **MPI_Get_count**(MPI_Status* status, MPI_Datatype datatype, int* count)
- **MPI_Recv**(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)

MAIN METHOD

Initialization:

We initialize MPI execution environment with the **MPI_Init**(int *argc, char ***argv) method. Then we initialize “rank” and “size” variables denoting the rank of the current processor and the total number of processors respectively with the **MPI_Comm_rank**(MPI_Comm communicator, int* rank) and **MPI_Comm_size**(MPI_Comm communicator, int* size) methods. Then we branch according to the rank.

Master processor:

- It reads the input file line by line and stores the words in the input file in a vector of type word.
- It splits the words (as equal as possible) and sends them to slaves.
- It receives pairs from slaves.
- It splits the pairs (as equal as possible) and sends them to slaves.
- It receives sorted pairs from slaves and merges these sublists into a single vector.
- It reduces pairs with the same words (they are consecutive) into a single pair by adding up their number of occurrences.
- It writes these reduced sorted pairs to the output file line by line.

Slave processors:

- They receive words from master, create pairs from these words (with counts 1) and send these pairs to master.
- They receive pairs from master, sort these pairs and send them to master.

Termination:

We terminate MPI execution environment with the **MPI_Finalize**() method and return 0.

EXAMPLES

IMPROVEMENTS and EXTENSIONS

All words having a fixed maximum length is the part that needs improvement in the project. I tried to use strings to overcome this problem but I could not pass vector of strings with MPI (because I could not know the size of string beforehand) thus I returned to fixed size char arrays. The strong point is that even with the large inputs this program runs seamlessly with enough number of processors (making use of parallel programming).

DIFFICULTIES ENCOUNTERED

Getting used to MPI is hard. I had so much difficulty while debugging the program. There were lots of messages passed and it was really hard to keep track of every one of them. The error messages of wrapper compiler are not very descriptive. After numerous segmentation faults, I eventually got the logic. Also passing vectors was not straightforward. After extensive research, I decided to transfer the vector as a char array because it was the simplest solution (simpler than creating a new MPI_Datatype from the structs, vector also consists of char arrays).

CONCLUSION

This project was a good opportunity to practice parallel programming. I learned how to use Open MPI and met with the parallel programming world. I am impressed that I created a decent parallel program with only 8 additional methods which is relatively easy.