CMPE300: Analysis of Algorithms
MPI PROGRAMMING PROJECT: DOCUMENTATION

Course ID: CMPE300
Course Title: Analysis of Algorithms
Students: Ali Alperen Sönmez, Ramazan Burak Sarıtaş
Student IDs: 2020400354, 2020400321
Project Title: MPI Programming Project
Type of the Project: Programming Project
Submission Date: 20.12.2022

Introduction

In this project, we were required to calculate the data for bigram language model using MPI framework. Message Passing Interface (MPI) is utilized to do our calculations in parallel. We used Open MPI implementation of MPI, Python programming language and mpi4py library of Python. We had 5 requirements to satisfy in this project, to briefly explain:

Requirement 1: Read the document file and distribute the data evenly to the worker processes.
Requirement 2: After receiving the data, worker process prints its rank and number of data it has. Then calculates the frequencies of bigrams and unigrams. If the merge_method argument is MASTER, they send their calculated data to master process.

Requirement 3: If the merge_method argument is WORKERS, each worker merges its data with the previous worker's data and sends the merged data to next worker. Last worker sends the merged data to master process.

Requirement 4: Master process calculates and prints the conditional probabilities of bigrams given in the test file.

Requirement 5: Writing a project report.

Program Interface

To run the program, you should have python, mpi4py and Open MPI installed in your computer. You should call the program from command line. An example program call is given below:

```
“mpiexec –n 5 python main.py ––input_file data/sample_text.txt
––merge_method MASTER ––test_file data/test.txt”
```

Where:

- -n flag stands for number of processes
- main.py is the source code file
- --input_file flag stands for the relative path of the data file
- --merge_method flag stands for the merge method explained in introduction section
- --test_file flag stands for the test file

Program Execution

The program takes input file indicated by –input_file flag in txt format. This file includes separated sentences.
First, master method distributes the data evenly to the workers, after that, each worker prints its rank and number of data it received.
Then, workers calculate frequencies of bigrams and unigrams in those sentences and merges the data according to the merge method.
Finally, program reads test file given in txt format and indicated by –test_file flag, calculates the conditional probabilities of the bigrams in test file, then, prints out the bigrams and their conditional probabilities.

Input and Output

Input: The program takes two separate files as an input:
1-) Data file given with –input_file flag. It is a text file having format:

```
<s> sentence </s>
<s> sentence </s>
…
```

2-) Test file containing bigrams, given with –test_file flag. It is a text file having format:

```
bigram word
bigram word
…
```

Output: The program output consists of two phases:
1-) After receiving the data, each worker prints its rank and number of sentences it has:

```
Rank 1 received 59108 sentences.
Rank 2 received 59108 sentences.
…
```

2-) After master calculated the conditional probabilities of bigrams in the test file, it prints the bigrams and their conditional probabilities:

```
pazar günü : 0.4462962962962963
pazartesi günü : 0.5966101694915255
…
```

Program Structure

Program starts by initializing MPI communication. Then, the master parses the command line arguments and sends the merge method to workers.

Functions read_file(), split_line():

After that, master reads the data in input file by using read_file() function. read_file() function reads the input file line by line, after reading each line, split_line() function splits the line into tokens and eliminates the leading and trailing tokens (<s> and </s>).

Function distribute_data():

Then, master distributes the data evenly to workers by using distribute_data() function. After receiving data, each worker prints its rank and number of sentences it got.

Function count_unigrams_bigrams():

Then, count_unigrams_bigrams() function traverses the sentences in the data of worker, calculates frequencies of unigrams and bigrams in each sentence and records them in a dictionary. Then returns the dictionary.

Functions merge_data_master(), merge_data_workers():

After calculations, if merge method is MASTER, program calls merge_data_master() function. It checks the rank of the process. If the process is a worker (rank!=0), the worker sends its dictionary to the master. If the process is master (rank==0), the master process receives dictionaries of the worker processes and merges them into a single dictionary.

If merge method is WORKERS, program calls merge_data_workers() function. It checks the rank of the process:

1-) If rank==1, first worker sends its data to next worker.

2-) If rank>1 but the process is not the last worker, the process receives the dictionary from previous worker, merges the dictionary with its dictionary and sends the merged dictionary to next process.

3-) If the process is the last worker, it merges its dictionary with the dictionary it received from the previous worker and sends the merged dictionary to the master.

Function compute_conditional_probability():

Finally, master process reads the test_file line by line. After reading each line, calculates the conditional probabilities of bigrams using compute_conditional_probability() function.
It takes the frequency dictionary, a bigram like "new technologies" and the unigram like "new" as arguments and returns the conditional probability of the bigram.

After the calculation, program prints the bigram and its conditional probability.

Design Decisions

1- Our program frequently uses if else blocks to discriminate the worker processes from the master process.
2- We used dictionaries as main data structure because they are quite time efficient when it comes to searching items.
3- Rather then holding dictionaries of workers in a list in master, we preferred to merge the dictionaries of workers into a dictionary in master.

Assumptions

While building the project, we assumed that:

1- Necessary input files used during the program are always present in the same directory with the source code file and are not corrupted.
2- The data provided in the input files are always valid and obeys the structure described in the project description file.
3- The number of worker processes is at least 1, hence input parameter -n is at least 2.
4- Master process is the process with rank 0.

Examples

With files input_file = sample_text.txt and test_file = test.txt provided along with the project description:

The output of the program for (number of processes) n = 5 is:

```
Rank 1 received 59108 sentences.
Rank 2 received 59108 sentences.
Rank 3 received 59109 sentences.
Rank 4 received 59109 sentences.
pazar günü : 0.4462962962962963
pazartesi günü : 0.5966101694915255
karar verecek : 0.010940919037199124
karar verdi : 0.13216630196936544
boğaziçi üniversitesi : 0.37272727272727274
bilkent üniversitesi : 0.2222222222222222
```

Similarly, the output for n = 2 is:

```
Rank 1 received 236434 sentences.
pazar günü : 0.4462962962962963
pazartesi günü : 0.5966101694915255
karar verecek : 0.010940919037199124
karar verdi : 0.13216630196936544
boğaziçi üniversitesi : 0.37272727272727274
bilkent üniversitesi : 0.2222222222222222
```

Number of worker processes is always n-1, master process having the rank 0. So the output displays number of received sentences of n-1 processes.

Possible Improvements

The efficiency of the program changes depending on the selected merge method for given number of data and number of worker processes. So, the program might select the best option for merge method based on the given initial conditions and data.

Difficulties Encountered

In the merge method WORKER, managing the data flow both in-between worker processes and worker processes and master process is quite challenging. Every process must receive and send the correct data, in correct order, in correct number. All requests of sending or receiving the data must be fulfilled. Otherwise, one unfulfilled receive request will make the program wait forever and the process will never be completed.

Conclusion

Executing complex and time-taking programs in a single process is quite inefficient. Having multiple worker processes to perform the same operation on different sections of the main data makes the program significantly efficient compared to a single process program.

Different options for collecting the resulting data from worker processes to the master process in the end are available, and the efficiency of them depends of the initial conditions and given data.