20.12.2022

CMPE300 Algorithm Analysis
MPI Programming Project Report

Volkan Öztürk & Arda Arslan
2019400033 2020400078

Computer Engineering Department
Engineering Faculty
Bogazici University

This documentation includes the following sections:

1) Introduction
2) Program Interface
3) Program Documentation
4) Program Execution
5) Input and Output
6) Program Structure
7) Examples
8) Improvements and Extensions
9) Difficulties Encountered
10) Conclusion
11) Appendix

Introduction

In this project we calculated the data for a bigram language model. We calculated the probabilities of the bigrams in the test_file. We have calculated the frequencies of the unigrams and the bigrams to calculate the probability. The calculations are made using the MPI framework.
-Bigram: consecutive sequence of 2 words.
-Unigram: consecutive sequence of one word.

Program Interface

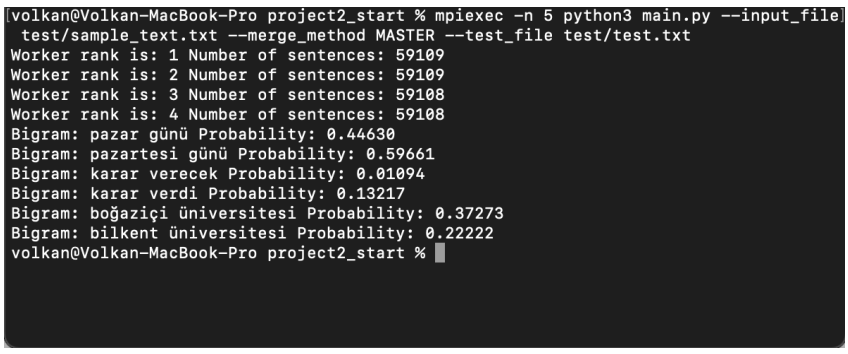
One should have python3 and Open MPI properly installed. (The installation guide for Open MPI is provided to student.) Then using the pip3 command 'pip3 install mpi4py' required library can be installed. User interacts with the program from terminal. An example program call is given below.

mpiexec -n <number_of_processors> python3 main.py –input_file <sample_file> --merge_method <MASTER or WORKERS> --test_file <test_file>

The program terminates without additional command.

Program Execution

- Types of the input
  <sample_file> includes the lines to be examined for the probabilities of the bigrams.
  <test_file> includes the bigrams those probabilities are to be calculated.
- Types of the output
  There is **no** output file. There are two types of prints on console. First group tells the information about the rank of the processor and its data size as number of lines assigned by the processor with rank 0. Second group tells the probabilities of the bigrams that are provided in the <test_file>.
- Functionality of the program
  Program calculates the probabilities of the bigrams.



Input and Output

Input file is a txt file contains data as lines. Lines start with special token <s> and ends with </s>. There is no output file. There are two types of prints on console. First group tells the information about the rank of the processor and its data size as number of lines assigned by the processor with rank 0. Second group tells the probabilities of the bigrams that are provided in the <test_file>. **(Probability is rounded to 5 decimal places.)**

Program Structure

The structure complies with the MPI framework. Processor ranked 0 (master) executes the **main** function. Other processors (workers) wait until the merge method is determined. After merge method is determined, processors either execute **method_master** function or **method_workers** function. Master processor waits the data according to the merge method. If method is master it waits the data from all other workers, otherwise it just waits the data from the processor with the highest rank. After receiving data, it merges the data if the method is master, if method is workers no additional operation is needed. Finally it calls the **calculate_bigram_probability** function to print the regarding probabilities of the bigrams that are read from the test_file. Each function details are given below.

- main(argv)
  Input file, merge method and the test file are determined with the arguments. The merge method information is sent to all workers. The data from input file is equally distributed to workers as lines. Receiving mechanism is implemented; if the method is master, master processor waits the data from all the workers, else if the method is workers it only waits from the processor with highest rank. After receiving data if the method is master the data are merged otherwise no additional operation is needed. Finally test_file is read to determine the bigrams whose probabilities are to be calculated, calculate_bigram_probability function is called with bigrams as list and the data as dictionary whose keys are the unigram/bigrams and whose values are the frequencies of the respective key. The program ends after this function returns.

- calculate_freq(dist_list)
  This function calculates the frequencies of the unigrams/bigrams that are in the elements of the dist_list argument as lines.

- method_master()
  Workers wait until the data had come. Then each worker prints its rank and the number of lines that is assigned to itself. Then calculate_freq function is called with the corresponding data. Return value of the calculate_freq function function is sent to the master by each worker.

- method_workers(caller_rank)
  Workers wait until the data had come. Then each worker prints its rank and the number of lines that is assigned to itself. If the processor is the rank with 1, it sends its data to processor 2. All workers whose rank is other than 1 or “highest” waits data from their preceding worker and sends the cumulative data (data calculated by the worker + data from the preceding worker) to the next worker. The processor with the highest rank merges the final data (data calculated by itself + cumulative data from all other workers) and sends it to the master processor.

- calculate_bigram_probability(biglist, data_dict)
  Each probability of bigram in biglist is calculated with the formula:
  P(<token1> <token2>) = Freq(<token1> <token2>) / Freq(<token1>)
  Frequencies are read from the data_dict.

Examples

In this example input file we have six lines consisted of some unigrams and bigrams. Program calculates the frequencies of each unigram and bigram. Lines are equally distributed to worker processors. First four lines of the console below shows the distribution numbers. Last line of the console below shows the probability of “CMPE 300” bigram. We can also see the probability from the input file. There are 4 different bigrams whose first token is “CMPE” and there are 5 instances of them 2 of which have the second token as “300”. Thus, the probability of bigram “CMPE 300” denoted P(“CMPE 300”) is 2/5 = 0.4.

```
Worker rank is: 1 Number of sentences: 2
Worker rank is: 2 Number of sentences: 2
Worker rank is: 3 Number of sentences: 1
Worker rank is: 4 Number of sentences: 1
Bigram: CMPE 300 Probability: 0.40000
```

Improvements and Extensions

We send and receive data with **send** and **recv** functions. As an improvement we can use **scatter** and **gather** for a better design and implementation. Note that this does not change the correctness of the program.

Difficulties Encountered

We found the task is to be straightforward and we did not face with any major difficulties.

Conclusion

The project is very informative for learning the MPI framework. The language model stated in the description was also interesting.

Appendix

Program source code:

```python
import sys, getopt
from mpi4py import MPI

comm = MPI.COMM_WORLD
world_size = comm.Get_size()
rank = comm.Get_rank()


def main(argv):  # Main function that processor 0 executes.
    input_file = ''  # Name of the input file.
    merge_method = ''  # Type of the merge method.
    test_file = ''  # Name of the test file.
    opts, args = getopt.getopt(argv,"",["input_file=","merge_method=","test_file="])  # To differentiate argument flags.
    for opt, arg in opts:
        if opt == "--input_file":
            input_file = arg
        elif opt == "--merge_method":
            merge_method = arg
        elif opt == "--test_file":
            test_file = arg
    method = merge_method
    for q in range(1, world_size):
        comm.send(method, dest=q)  # Processor 0 sends the method type to worker processors.

    input_open = open(input_file, 'r', encoding="utf-8")  # Opens the input file with utf-8 encoding to avoid errors of using characters that is not in the English alphabet.
    distribution_dict = dict()  # Dictionary that keeps the lines to be distributed. Key is the rank (or id) of the processor, value is the list of lines that is assigned to the regarding processor.
    for v in range(1, world_size):
        distribution_dict[v] = list()  # Initialize the lists of all processors.
    counter = 0  # Counts the number of lines, facilitates the equal distribution with the modulo operator.
    for line in input_open:  # For loop equally distributes lines to the worker processors.
        l = distribution_dict[(counter % (world_size - 1)) + 1]
        l.append(line)
        distribution_dict[(counter % (world_size - 1)) + 1] = l
        counter += 1
    for i in range(1, world_size):  # For loop that sends the data to the worker processors. Data is a list consisting of the lines to be examined.
        comm.send(distribution_dict[i], dest=i)

    data_dict = dict()  # Dictionary for keeping the frequency of unigrams and bigrams.
    if method == 'MASTER':  # If the method is MASTER, then processor 0 seperately receives the data from all other worker processors and adds the frequencies to main dictionary, named data_dict.
        for i in range(1, world_size):  # For loop to add seperate data that are sent by the workers to the processor 0.
            data = comm.recv(source=i)
            for unit in data.keys():  # For loop to add the given frequencies from the data that is sent by the regarding worker, to the main dictionary. Finally we have the dictionary that has the cumulative frequencies.
                data_dict[unit] = data_dict.get(unit, 0) + data[unit]
    else:  # If the method is WORKERS, then processor 0 only receives data from processor world_size - 1, and the contents are loaded to the main dictionary, named data_dict.
        data = comm.recv(source=(world_size-1))
        for unit in data.keys():
            data_dict[unit] = data_dict.get(unit, 0) + data[unit]

    input_test_open = open(test_file, 'r', encoding="utf-8")  # Opens the test file with utf-8 encoding to avoid errors of using characters that is not in the English alphabet.
    biglist = list()  # List that keeps the bigrams from the test file.
    for line in input_test_open:  # For loop that adds the bigrams to the biglist.
        biglist.append(line.strip())
    calculate_bigram_probability(biglist, data_dict)  # Processor finally calls this function to calculate the probabilities of the given bigrams.

def calculate_freq(dist_list):  # Function to calculate the frequencies of the unigrams and bigrams from all the lines that the regarding processor is assigned to, by processor 0.
    return_dict = dict()  # Dictionary that keeps the frequencies of the unigrams and bigrams. Key is the unigram/bigram, value is the count.
    for line in dist_list:  # For loop that iterates through the lines that the regarding processor is assigned to.
        token_list = line.split()  # Splits all the tokens of the line.
        for k in range(len(token_list) - 1):  # For loop to iterate the tokens. Note that final token is seperately handled to avoid 'index out of bound error' that would happen if we try to reach the index len(token_list).
            bigram = token_list[k] + ' ' + token_list[k + 1]  # Creates the bigram with the given index.
            return_dict[token_list[k]] = return_dict.get(token_list[k], 0) + 1  # Increments 1, the count of the regarding unigram.
            return_dict[bigram] = return_dict.get(bigram, 0) + 1 # Increments 1, the count of the regarding bigram.
        return_dict[token_list[-1]] = return_dict.get(token_list[-1], 0)  # Increments 1, the count of the final token.
    return return_dict

def method_master():  # Function to use when the method is MASTER.
    data = comm.recv(source=0)  # All workers receives data from the processor 0.
    print("Worker rank is:", rank, "Number of sentences:", len(data))  # Print requirement in Requirement 2.
    data_dict = calculate_freq(data)  # All workers calls this function to calculate the frequencies of bigrams and the unigrams from the data that is assigned by processor 0.
    comm.send(data_dict, dest=0)  # All workers send their respective data to the processor 0, with the dictionary type. Key is the unigram/bigram, value is the frequency / count.
    return

def method_workers(caller_rank):  # Function to use when the method is WORKERS.
    data = comm.recv(source=0)  # All workers receives data from the processor 0.
    print("Worker rank is:", rank, "Number of sentences:", len(data))  # Print requirement in Requirement 2.
    data_dict = calculate_freq(data)  # All workers calls this function to calculate the frequencies of bigrams and the unigrams from the data that is assigned by processor 0.
    if caller_rank == 1:  # Worker that has the rank 1 receives no data since it is the first worker, So it just sends its respective data to the next processor, namely 2.
        comm.send(data_dict, dest=(caller_rank + 1))
    elif caller_rank != (world_size - 1):  # Workers that have the rank (1 < rank < world_size - 1) receives the data from their preceding processor in the form of the dictionary.
        data_dict2 = comm.recv(source=(caller_rank -1))  # Key is the unigram/bigram, value is the count.
        for key in data_dict2.keys():  # Workers adds the data from its preceding worker to its own data in order to have the data cumulatively.
            data_dict[key] = data_dict.get(key, 0) + data_dict2[key]
        comm.send(data_dict, dest=(caller_rank +1))  # Worker send the cumulative data to the next worker.
    else:  # Worker that has the rank world_size - 1 receives the data from its preceding processor in the form of the dictionary, and sends the final cumulative data to the processor 0.
        data_dict2 = comm.recv(source=(caller_rank -1))   # Key is the unigram/bigram, value is the count.
        for key in data_dict2.keys():
            data_dict[key] = data_dict.get(key, 0) + data_dict2[key]
        comm.send(data_dict, 0)  # Worker that has the rank world_size - 1 sends the final data in the form of the dictionary to the processor 0.
    return

def calculate_bigram_probability(biglist, data_dict):  # Function to calculate the probabilities of the bigrams in the biglist from the frequencies in the data_dict.
    for big in biglist:  # For loop to iterate the bigrams in the biglist.
        bigw = big.split()  # Seperates the bigram tokens in order to find first token.
        firstw = bigw[0]
        probability = data_dict.get(big, 0) / data_dict[firstw]  # Probability of the given bigram is = (Freq(<bigram>) / Freq(<first token of the bigram>)).
        print("Bigram:", big, "Probability: %.5f" % probability)  # The probability is printed with 5 digit precision after point.
    return

if rank == 0:  # Processor 0, MASTER, calls the main function.
    if __name__ == "__main__":
        main(sys.argv[1:])
else:
    method = comm.recv(source=0)  # Workers wait until the merge method is determined by the processor 0.
    if method == 'MASTER':  # If the method is MASTER, workers call the method_master function, otherwise they call method_workers function.
        method_master()
    else:
        method_workers(rank)
```