Project 2

MPI Programming on Bigram Language Model

CmpE 300, Analysis of Algorithms , Fall 2022

Bahadır Gezer - 2020400039
Muhammet Batuhan Ilhan - 2019400243

20 December 2022

Contents

1 Introduction

The N-Gram Language Model project is a parallel computing project that aims to calculate the probability of a given sequence of words occurring in a language. Specifically, this project focuses on implementing a bigram language model, where the probability of a word is predicted based on the previous word in the sequence.

This project utilizes the MPI (Message Passing Interface) framework to perform the calculations in parallel, utilizing the built-in data structures of the programming language. The use of parallel computing allows for more efficient and faster calculation of the language model probabilities. In our implementation we use the master-slave architecture for the MPI. Specifically, if there communication size is n we use one master thread and $n-1$ slave threads.

1.1 Definitions

Unigram: An unigram is a single word in a sequence. In a unigram language model, the probability of a word is calculated based on its individual frequency in the language.

Bigram: A bigram is a pair of words in a sequence. In a bigram language model, the probability of a word is calculated based on the previous word in the sequence.

Master Thread: In a parallel computing program, the master thread is responsible for distributing tasks to the slave threads and coordinating their execution.

Slave Thread: In a parallel computing program, a slave thread is a worker thread that performs a specific task assigned by the master thread. There may be multiple slave threads in a program, each working on a different task.

2 Program Interface

To start the N-Gram Language Model program, you will need to have MPI (Message Passing Interface) installed on your system and set up in your programming environment. Once you have MPI set up, you can activate the programming environment and start the program from the terminal on the `src` directory by entering the following command and three command line flags.

```
mpiexec -n <number_of_threads> python main.py
```

This command will start the program with the `number_of_threads` total threads. The three required flags are `--input_file`, `--test_file`, and `--merge_method`. The values after these flags will be used inside of the program to find the required information. One of which is the corpus text, it is in the input file. The bigram tests for the language model is in the test file. Lastly the merge method, which is used determines how the calculated data is passed through the threads. There are only two merge methods implemented in this program, the first one has a value of "MASTER" and the other is "WORKERS".

To terminate the program, simply allow it to run to completion. The program will terminate on its own once all calculations have been completed.

Please note that the specific details of the communication steps between the user and the program will be explained in the following sections.

3 Program Execution

The N-Gram Language Model program uses the MPI (Message Passing Interface) framework to perform calculations in parallel. It consists of a master thread and multiple slave threads, with the master thread responsible for distributing the input data to the slave threads and coordinating their execution.

1. **Data distribution:** The master thread reads in the input document file and evenly distributes the data to the slave threads. The slave threads each receive a portion of the data and print their rank and the number of sentences they have received.

2. **Bigram and unigram counting:** Each slave thread counts the bigrams and unigrams in the sentences it received. The slave threads perform this calculation in parallel.

3. **Data merging:** The slave threads send their calculated bigram and unigram counts back to the master thread, which merges the data by summing the counts of the same bigrams and unigrams. There are two options for merging the data, specified by the "–merge_method" flag:

    - **MASTER:** After each slave thread has finished calculating the data, they send it to the master thread. The master thread is responsible for merging the data by summing the counts of the same bigrams and unigrams.
    - **WORKERS:** Instead of passing the calculated data to the master thread, each slave thread gathers the data from the previous slave thread, merges that data with its own data, and passes it to the next slave thread. The last slave thread passes the final data to the master thread.

4. **Conditional probability calculation:** The master thread uses the merged bigram and unigram counts to calculate the conditional probabilities for each pair of words in the test data file.

5. **Program termination:** The program terminates once all calculations have been completed and the output has been generated.

3.1 Input

- Preprocessed text document containing a single sentence per line, with special tokens "<s>" (start of sentence) and "</s>" (end of sentence) in the beginning and end. This file is read from the command line argument "–input_file".
- Test data file containing pairs of words separated by a single space, representing conditional probabilities to be calculated. This file is read from the command line argument "–test_file".

3.2 Output

The output is passed from the standard output console.

- The rank and number of sentences received by each slave thread.
- Calculated conditional probabilities for each pair of words in the test data file

4 Program Structure

The N-Gram Language Model program consists of two Python files:

- `src/main.py`: This file handles the communication between the master and slave threads using the `mpi4py` package. It includes a `main()` function that is called when the program is executed.
- `src/model/model.py`: This file contains two classes: `Master` and `Slave`. The `Master` class is used by the `main.py` file to handle the distribution of data to the slave threads and the merging of data from the slave threads. The `Slave` class is used by the `main.py` file to represent each

slave thread, which is responsible for counting the bigrams and unigrams in the data it receives and returning the results to the master thread.

In the `main()` function of `main.py`, the `MPI.Intracomm` object is used to set up the communication between the master and slave threads. The size and rank of the threads are determined using the size and rank properties of the `MPI.Intracomm` object. The `merge_method` variable is set based on the "–merge_method" flag passed in the program execution command.

If the size of the threads is 1, a `ValueError` is raised as there must be at least one slave thread. If the rank of the current thread is 0, it is the master thread and it executes the code for the `Master` class. The `Master` class reads in the input data, evenly distributes it to the slave threads, and sends the data to the slave threads using the `send()` method of the `MPI.Intracomm` object.

The `merge_method` variable is then checked to determine the workflow for merging the data from the slave threads. If the value is "MASTER", the master thread receives the data from the slave threads using the `recv()` method of the `MPI.Intracomm` object and merges the data using the `merge()` method of the `Master` class. If the value is "WORKERS", the master thread waits to receive the merged data from the last slave thread using the `recv()` method of the `MPI.Intracomm` object. Once all the data has been merged, the master thread displays the results using the `display_results()` method of the `Master` class.

If the rank of the current thread is not 0, it is a slave thread and it executes the code for the `Slave` class. The slave thread receives the data from the master thread using the `recv()` method of the `MPI.Intracomm` object and counts the bigrams and unigrams in the data using the `count_ngrams()` method of the `Slave` class.

The slave thread then sends the results back to the master thread based on the value of the `merge_method` variable. If the value is "MASTER", the

slave thread sends the results directly to the master thread using the `send()` method of the `MPI.Intracomm` object. If the value is "WORKERS", the slave thread receives the merged data from the previous slave thread using the `recv()` method of the `MPI.Intracomm` object, merges the data with its own results using the `merge()` method of the `Slave` class, and sends the merged data to the next slave thread or the master thread if it is the last slave thread.

Once all the data has been processed and the results have been sent back to the master thread, the `main()` function returns and the program terminates.

Additionally, it should be added that every communication in this program is blocking communication. So it is imperative that every thread does its part correctly.

In summary, the N-Gram Language Model program uses the `mpi4py` package to set up a master-slave architecture for parallel processing of data. The `Master` class is responsible for distributing the data to the slave threads and merging the results from the slave threads, while the `Slave` class is responsible for counting the bigrams and unigrams in the data it receives and returning the results to the master thread. There are two options for merging the data: the MASTER method, where the data is merged by the master thread, and the WORKERS method, where the data is merged sequentially between the slave threads before being sent to the master thread. The results are then displayed by the master thread.

5 Examples

Below is a sample output of the program. It uses the default corpus and test files provided by the course. There are a total of 4 threads in this program.

```
Rank: 1, Sentences: 78812
Rank: 2, Sentences: 78811
```

```
Rank: 3, Sentences: 78811
pazar günü -> 0.4463
pazartesi günü -> 0.5966
karar verecek -> 0.0109
karar verdi -> 0.1322
boğaziçi üniversitesi -> 0.3727
bilkent üniversitesi -> 0.2222
```

6 Improvements and Extensions

There are several potential improvements and extensions that can be made to the N-Gram Language Model program. One improvement would be to expand the program to support N-grams beyond just bigrams, allowing for more flexibility and accuracy in language modeling tasks.

Another improvement would be to add additional options for merging the data, beyond just the MASTER and WORKERS methods currently implemented. This could include methods such as a tree-based approach, where the data is merged in a tree structure rather than a linear sequence.

Additionally, the program is currently limited to using a preprocessed text corpus, which may not always be available or suitable for a particular language modeling task. It would be useful to extend the program to allow for the creation of a corpus from raw text data, potentially using additional libraries for preprocessing and tokenization.

7 Difficulties Encountered

One of the main difficulties encountered in this project was the implementation of the MPI framework and the use of message sending operations for parallel processing. As this was the first time we were working with the MPI framework, there was a learning curve in understanding how to set up the

master-slave architecture and communicate between the threads.

In addition, using the message sending operations for data transfer between the threads introduced the potential for blocking communication, which could have caused the program to become stuck and not finish executing. To prevent this, it was important to carefully design the communication workflow and use the appropriate message sending and receiving operations.

8 Conclusion

In this project, we implemented an N-Gram Language Model program that calculates the bigram and unigram frequencies and conditional probabilities from a given input text file using the MPI framework. Overall, the program was successful in calculating the bigram and unigram frequencies and conditional probabilities in parallel, making use of the MPI framework to improve the efficiency and speed of the calculations. The program was able to handle large input files and process the data efficiently, making it a useful tool for language modeling tasks.

A Source Code

A.1 main.py

```
"""
    Student Name: Bahadır Gezer / Muhammet Batuhan Ilhan
    Student Number: 2020400039 / 2019400243
    Compile Status: Compiling
    Program Status: Working
    Notes: The program is working as expected.
        Just run the mpiexec command using main.py with the required
↪   arguments inside the src/ folder.
"""
import sys
```

```python
from mpi4py import MPI

import model.model as model

'''
    Use the mpi framework to distribute the work across multiple workers.
↪   It will be a master salve architecture.
    The master will be responsible for reading the csv file and
↪   distributing the work to the slaves.
    The slaves will be responsible for counting the bigrams and returning
↪   the results to the master.
    The master will then merge the results and return the final result to
↪   the user.
'''


def main() -> None:
    comm: MPI.Intracomm = MPI.COMM_WORLD
    # the master will be rank 0, so we need to subtract 1 from the size to
    ↪  get the number of workers
    size: int = comm.size
    rank: int = comm.rank
    merge_method: str = model.merge_method(sys.argv)

    if size == 1:
        raise ValueError("There must be at least one slave.")
    if rank == 0:
        # master
        master: model.Master = model.Master(sys.argv, size - 1)
        # send the distributed data to the workers
        for worker in range(1, size):
            comm.send(master.data[worker - 1], dest=worker)
        # receive the results from the workers based on the merge method
        if merge_method == "MASTER":
            # merge the results on the master
            for _ in range(1, size):
```

```
                calculated: dict = comm.recv()
                master.merge(calculated)
        # master will merge the result from the last worker
        elif merge_method == "WORKERS":
            calculated: dict = comm.recv(source=size - 1)
            master.merge(calculated)
        else:
            raise ValueError("Invalid merge method.")
        # display the results
        master.display_results()
    else:
        # slaves
        # receive the data from the master
        work: list[str] = comm.recv(source=0)
        slave: model.Slave = model.Slave(rank, work)
        slave.count_ngrams()
        # send the results back to the master based on the merge method
        if merge_method == "MASTER":
            # send the result to the master directly
            comm.send(slave.freqs, dest=0)
        elif merge_method == "WORKERS":
            # receive the result from the last worker, merge the results
            ↪  and send it to the master
            # the first worker does not receive anything
            if rank != 1:
                calculated: dict = comm.recv(source=rank - 1)
                slave.merge(calculated)
            # last worker sends the result to the master
            dest_rank = rank + 1 if rank + 1 < size else 0
            comm.send(slave.freqs, dest=dest_rank)
        else:
            raise ValueError("Invalid merge method.")
    return None


# mpiexec -n 4 python main.py --input_file ../resources/sample_text.txt
↪  --merge_method MASTER --test_file ../resources/test.txt
```

```
if __name__ == '__main__':
    main()
```

A.2 model/model.py

```
"""
    Student Name: Bahadır Gezer / Muhammet Batuhan Ilhan
    Student Number: 2020400039 / 2019400243
    Compile Status: Compiling
    Program Status: Working
    Notes: The program is working as expected.
        Just run the mpiexec command using main.py with the required
↪   arguments inside the src/ folder.
"""
import sys


class Master:
    def __init__(self, argv: list[str], size: int):
        self.merge_method: str = ""
        self.test_file: str = ""
        self.input_file: str = ""
        self.data: list[list[str]] = []
        self.tests: list[str] = []
        self.freqs: dict = dict()
        self.parse_args(argv)
        self.divide_work(self.parse_corpus(), size)
        self.parse_test()

    def parse_args(self, argv: list[str]) -> None:
        # flag arguments can be in any order
        for i in range(len(sys.argv)):
            if argv[i] == "--test_file":
                self.test_file = sys.argv[i + 1]
            if argv[i] == "--input_file":
```

```
            self.input_file = sys.argv[i + 1]
        if argv[i] == "--merge_method":
            self.merge_method = sys.argv[i + 1]

    if self.input_file is None or self.test_file is None or
    ↪  self.merge_method is None:
        # raise an error for missing arguments
        raise ValueError("Missing program arguments.")

def parse_corpus(self) -> list[str]:
    with open(self.input_file, "r") as f:
        corpus: list[str] = f.readlines()
    return corpus

def divide_work(self, work: list[str], workers: int) ->
↪  list[list[str]]:
    # divide the work equally among the workers,
    # if there are any left over, distribute them one at a time until
    ↪  all work is distributed
    work_per_worker: int = len(work) // workers
    for i in range(workers):
        start = i * work_per_worker
        end = (i + 1) * work_per_worker
        self.data.append(work[start:end])
    leftovers = len(work) % workers
    for i in range(leftovers):
        self.data[i].append(work[-1 - i])
    # print the execution time in seconds to 4 decimal places
    return self.data

def parse_test(self) -> list[str]:
    with open(self.test_file, "r") as f:
        # split and strip the test file line by line
        lines = [line.strip() for line in f.readlines()]
        for line in lines:
            self.tests.append(" ".join(line.split()))
    return self.tests
```

```
    def display_results(self) -> None:
        # for each test
        for test in self.tests:
            freq_first_word: int = self.freqs.get(test.split()[0], 0)
            freq_test: int = self.freqs.get(test, 0)
            if freq_first_word == 0:
                print(f"{test} -> {0}")
            else:
                # probability of the bigram
                prob: float = freq_test / freq_first_word
                # format the probability to 4 decimal places
                print(f"{test} -> {format(prob, '.4f')}")

    def merge(self, calculated_ngrams: dict) -> None:
        self.freqs = {k: self.freqs.get(k, 0) + calculated_ngrams.get(k,
        ↪  0)
                      for k in set(self.freqs).union(calculated_ngrams)}


class Slave:
    def __init__(self, rank: int, work: list[str]):
        self.rank: int = rank
        self.work: list[str] = work
        self.freqs: dict = dict()
        print(f"Rank: {rank}, Sentences: {len(work)}")

    def count_ngrams(self) -> dict:
        # count the ngrams
        for sentence in self.work:  # unigrams
            for word in sentence.split():
                self.freqs[word] = self.freqs.get(word, 0) + 1
        for sentence in self.work:  # bigrams
            words = sentence.split()
            for i in range(len(words) - 1):
                bigram = words[i] + " " + words[i + 1]
                self.freqs[bigram] = self.freqs.get(bigram, 0) + 1
```

```
        return self.freqs

    def merge(self, calculated_ngrams: dict) -> None:
        self.freqs = {k: self.freqs.get(k, 0) + calculated_ngrams.get(k,
        ↪  0)
                      for k in set(self.freqs).union(calculated_ngrams)}


def merge_method(argv: list[str]) -> str:
    # return the merge method
    for i in range(len(sys.argv)):
        if argv[i] == "--merge_method":
            return sys.argv[i + 1]
    raise ValueError("Missing program arguments.")
```