CMPE300 : MPI Programming Project

YİĞİT SARIOĞLU - 2022400354

GÜNEY YÜKSEL -2018400102

Boğaziçi University

Submission Date : 20.12.2022 Tuesday

Project Type : Parallel Programming Project with Python

Notes

Introduction

Unigrams and bigrams are types of n-grams, which are contiguous sequences of n items (usually words or characters) in a text. Unigrams are single words, while bigrams are pairs of words. In language processing, n-grams are often used as a basic building block for various tasks, such as language modeling, text classification, and information retrieval. For example, a language model predicts the probability of a sequence of words occurring in a language, and it can be trained on a large dataset of unigrams, bigrams, and higher-order n-grams.

In this project, we explore the use of the Message Passing Interface (MPI) for parallelizing the computation of bigrams and unigrams in a collection of sentences. MPI is a standardized and widely used library for writing parallel programs in C, C++, and Fortran, but it can also be used in Python through the mpi4py package. By leveraging the power of multiple processors or computing nodes, MPI can significantly speed up the execution of certain types of tasks, such as counting the occurrences of n-grams in a large dataset.

The task of counting bigrams and unigrams can be easily parallelized by dividing the input sentences among the available processes and having each process count the n-grams in its assigned portion. The results can then be gathered and combined by the main process to obtain the final counts. There will be two types of workflows for merging the calculated data. (master or workers)

The program is supposed to work in a master slave/worker architecture. There will be P processes where the rank of the master process is zero and the ranks of the worker processes are positive. This means that there will be P-1 workers. The master node will be responsible for

reading and distributing the input data evenly to the processes. The workers are responsible for calculating the bigram data in parallel. At the end, the data will be gathered in the master node.

In this project, the master process will compute the conditional probabilities of the bigrams that are read from the input test file (as required by requirement 4). We calculated the conditional probabilities of "boğaziçi üniversitesi" as follows :

P (üniversitesi | boğaziçi) = Freq (boğaziçi üniversitesi) / Freq (boğaziçi)

P (boğaziçi üniversitesi) = P (boğaziçi) * P (üniversitesi | boğaziçi)

The above equations are our assumptions when calculating conditional probability in our project.

Program Interface

- You must have Python installed on your computer.
- You need to install Open MPI. (https://www.open-mpi.org/)
- mpi4py (MPI for Python Package) also should be installed

Run the program from the terminal with this command:

- "mpiexec -n <process_number> python main.py --input_file <input_file_location> --merge_method <merge_method> --test_file <test_file_location>".
- Users can specify the number of processes
- Users can specify merging method as MASTER or WORKERS.
- Users can specify the input_file and test_file.

Program Execution

This program is written in Python with an MPI environment. It calculates the bigram and unigram numbers in parallel and calculates the conditional probability of the given bigram.

1. When user executes the command :

   -mpiexec -n 5 python3 main.py --input_file data/sample_text.txt --test_file data/test.txt --merge_method MASTER

   program should print how many sentences each worker process received and conditional probability of each sentence in the test_file document.

```
yigit@yigit:~/Documents/cmpe300$ mpiexec -n 5 python3 main.py --input_file data/sample_text.txt --test_file data/test.txt  --merge_method MASTER
Process 1 has received 59109 sentences
Process 2 has received 59109 sentences
Process 3 has received 59109 sentences
Process 4 has received 59108 sentences
(MASTER METHOD)Conditional probability of pazar günü is 0.4462962962962963
(MASTER METHOD)Conditional probability of pazartesi günü is 0.5966101694915255
(MASTER METHOD)Conditional probability of karar verecek is 0.010940919037199124
(MASTER METHOD)Conditional probability of karar verdi is 0.13216630196936544
(MASTER METHOD)Conditional probability of boğaziçi üniversitesi is 0.37272727272727274
(MASTER METHOD)Conditional probability of bilkent üniversitesi is 0.2222222222222222
```

2. When user executes the command :

   -mpiexec -n 5 python3 main.py --input_file data/sample_text.txt --test_file data/test.txt --merge_method WORKERS

   program should print how many sentences each worker process received and conditional probability of each sentence in the test_file document.

```
yigit@yigit:~/Documents/cmpe300$ mpiexec -n 5 python3 main.py --input_file data/sample_text.txt --test_file data/test.txt  --merge_method WORKERS
Process 1 has received 59109 sentences
Process 2 has received 59109 sentences
Process 3 has received 59109 sentences
Process 4 has received 59108 sentences
(WORKER METHOD) Conditional probability of pazar günü is 0.4462962962962963
(WORKER METHOD) Conditional probability of pazartesi günü is 0.5966101694915255
(WORKER METHOD) Conditional probability of karar verecek is 0.010940919037199124
(WORKER METHOD) Conditional probability of karar verdi is 0.13216630196936544
(WORKER METHOD) Conditional probability of boğaziçi üniversitesi is 0.37272727272727274
(WORKER METHOD) Conditional probability of bilkent üniversitesi is 0.2222222222222222
```

Input and Output

Users will run the program with the command “mpiexec -n <process_number> python main.py --input_file <imput_file_location> --merge_method <merge_method> --test_file <test_file_location>”. <process_number> represents the number of processors requested to run this program. Using a greater number than the actual number of processors in the computer may create a problem in some computers. <input_file_location> represents the exact location of the file in the computer. <merge_method> represents the wanted method of merging in the process. It should be “MASTER” or “WORKERS”. If it’s not one of those options, the user will see a

message on the terminal stating that that input is not correct and it should be one of the two methods.<test_file_location> represents the exact location of the test file.

After the proper command is submitted, The program will print 2 things. First, it will print the number of sentences distributed to each worker. After those print statements' execution, the program will print the probability of the wanted test cases.

Program Structure

- The program first takes the arguments from the terminal and assigns them to the variables. You can give the arguments with certain keywords from the terminal. (--input_file , --merge_method , --test_file) . argparse library is used in python for this feature.
- The MPI environment was created with MPI.COMM_WORLD. Number of processors and rank of the processors found via this method.
- Which processor is working is decided according to whether the rank is 0 or not. If the rank is 0, the master processor is considered to be running and its work is done. If the rank is not equal to 0, then the worker processors are assumed to be running and their work is done. These operations are handled simply with the help of if else blocks.
- Bigram and unigrams are kept in python dictionaries named bigram_counts and unigram_counts.
- Firstly, the master process reads and distributes the input data (--input_file) evenly to the worker processes. This is done via the comm.send and comm.recv methods. The worker processes receive the sentences from the master process and prints the number of sentences received.

- Then, bigram and unigram counts are made according to the merge method using if else blocks in the code. This will be done by all the worker processes in parallel. Then, the bigram and unigram counts of the workers will be merged by summing the counts of the same bigrams and unigrams.
- If MASTER (merge method) isselected as the merging method, each worker sends its data to the master. Each worker process splits sentences into words and puts them in a dictionary named bigram_data. If data exists before, its value is taken and incremented by 1.
- If WORKER (merge method) is selected as the merge method, each worker process counts the number of bigrams and unigrams simultaneously. And the first process passes data to the second, and the second to the third, in this way it goes all the way to the last processor. At the end, the last processor sends the dictionaries named bigram_counts and unigram_counts to the master process.
- Then the master processor receives the data from the worker processors with the command comm.recv. It also takes the test file and calculates the conditional probability for each sentence.
- We did the conditional probability calculation as in the following example:

  P (üniversitesi | boğaziçi) = Freq (boğaziçi üniversitesi) / Freq (boğaziçi)

  P (boğaziçi üniversitesi) = P (boğaziçi) * P (üniversitesi | boğaziçi)

Examples

There are 2 main differences in executions of the program. Which is the merge method. If the merge method is “WORKERS”, the user will see the print statements of the workers in a slower way. Because the sentences distributed to the workers with lower ranks have a longer road to reach the master.

```
Process 1 has received 59109 sentences
Process 1 has received worker merging method
Process 2 has received 59109 sentences
Process 2 has received worker merging method
Process 3 has received 59109 sentences
Process 3 has received worker merging method
Process 4 has received 59108 sentences
Process 4 has received worker merging method
```

approximately 6.28 seconds for the whole execution

If the merge method is “MASTER”, the user will see the print statements much faster because all workers will process their sentences simultaneously.

```
Process 1 has received 59109 sentences
Process 2 has received 59109 sentences
Process 3 has received 59109 sentences
Process 4 has received 59108 sentences
```

approximately 3.12 seconds for the whole execution.

After the print statements, the user will see another print statement stating the probability of the wanted bigrams in the sample.

```
(MASTER METHOD)Conditional probability of pazar günü is 0.4462962962962963
(MASTER METHOD)Conditional probability of pazartesi günü is 0.5966101694915255
(MASTER METHOD)Conditional probability of karar verecek is 0.010940919037199124
(MASTER METHOD)Conditional probability of karar verdi is 0.13216630196936544
(MASTER METHOD)Conditional probability of boğaziçi üniversitesi is 0.37272727272727274
(MASTER METHOD)Conditional probability of bilkent üniversitesi is 0.2222222222222222
```

```
(WORKER METHOD) Conditional probability of pazar günü is 0.4462962962962963
(WORKER METHOD) Conditional probability of pazartesi günü is 0.5966101694915255
(WORKER METHOD) Conditional probability of karar verecek is 0.010940919037199124
(WORKER METHOD) Conditional probability of karar verdi is 0.13216630196936544
(WORKER METHOD) Conditional probability of boğaziçi üniversitesi is 0.37272727272727274
(WORKER METHOD) Conditional probability of bilkent üniversitesi is 0.2222222222222222
```

Improvements and Extensions

One of the first things to notice about this program is that the master method works much faster than the worker method, which is caused by some sentences having a much longer path than others. To be more specific, the sentences distributed to the lower ranked workers will get sent to the upper ranked workers and only then it will get sent to the master. To improve this, the user can be encouraged more to select the master route or workers method can be improved by altering the distribution function. The function distributes in a standard normal distribution type where all workers have the same number of sentences with a difference of 1 sentence. If this distribution is altered in a way such that the higher ranked workers have slightly more sentences than the lower ranked ones, the overall path will become shorter and the overall speed will get better. But despite this improvement the master methods will still work faster.

Difficulties Encountered

Coding in an environment which works in a synchronous way is something we don't have a lot of experience with. So the thinking process for some of the algorithms was challenging. In that sense, the difficulty of the project was extremely well-adjusted. If it were significantly harder, understanding how mpi operates and shaping it so that it will reach the wanted outcomes may become too much to handle.

Conclusion

In conclusion, MPI programming with Python is an effective way to calculate the conditional probabilities of bigrams and unigrams of sentences in parallel. By using the MPI

library, we are able to distribute the workload across multiple compute nodes, allowing us to take advantage of parallel processing to speed up the computation. This is especially useful for large datasets where the calculation of conditional probabilities can take a significant amount of time. Overall, MPI programming with Python is a powerful tool for optimizing the performance of natural language processing tasks.

We also worked with different types of programming/parallelization methods in our project. In the merging of data with the workers merge method, each worker sequentially transmits the data to the next worker process. In the master method, each worker made the calculation and sent it directly to the master process. It may also depend on the code written, but the master method was found to be faster.

Appendices

```
#Student Name:
#Student Number:
#Compile Status: Compiling
#Program Status: Working

import argparse
from mpi4py import MPI

parser = argparse.ArgumentParser()
parser.add_argument("--input_file", type=str, required=True)
parser.add_argument("--merge_method", type=str, required=True)
parser.add_argument("--test_file", type=str, required=True)

# get it
args = parser.parse_args()


# Read the input file from the command line arguments
input_file = args.input_file

# Read the test file from the command line arguments
```

```
test_file = args.test_file

# Read the merge method from the command line arguments
merge_method = args.merge_method


# Initialize the MPI environment
comm = MPI.COMM_WORLD

# Get the rank of the current process
rank = comm.rank

# Get the total number of processes
num_procs = comm.size

bigram_data = {}
unigram_data = {}

bigram_counts ={}
unigram_counts = {}


# Dictionary to store the conditional probabilities of the bigrams
probs = {}




# The master process is responsible for reading and distributing the input data evenly
# to the worker processes.
if rank == 0:
    # Read the input file and split it into sentences
    with open(input_file, "r") as f:
        sentences = f.read().split("\n")

    # Compute the number of sentences per process
    # sentences_per_process = len(sentences) // (num_procs-1)


    # Distribute the sentences evenly to the worker processes
    # First creating a sentence list then sending it to the proper workers
    sentence_lists = []
    for i in range(num_procs):
        sentence_lists.append([])
```

```
    for i in range(len(sentences)):
        sentence_lists[i % (num_procs-1)].append(sentences[i])

    for i in range(num_procs-1):
        comm.send(sentence_lists[i], dest = i+1)


else:
    # The worker processes receive the sentences from the master process
    sentences = comm.recv(source=0)

    print("Process {} has received {} sentences".format(rank, len(sentences)))



    if merge_method=="MASTER" :

         for sentence in sentences:
            # Split the sentence into words
            words = sentence.split(" ")

            # Remove the first and last word.

            # Count the frequencies of the bigrams and unigrams in the sentence
            for i in range(len(words)-1 ):
                bigram = (words[i], words[i+1])
                #abc def bigram = ("abc", "def"), "abc def"
                bigram_data[bigram] = bigram_data.get(bigram, 0) + 1

                unigram = words[i]
                bigram_data[unigram] = bigram_data.get(unigram, 0) + 1


         comm.send(bigram_data, dest=0)


    elif merge_method == "WORKERS":
        # bigram_counts and unigram_counts are dictionaries containing the bigram and unigram
counts for each worker process

        # Gather the bigram and unigram counts from the previous worker (if any)
        print("Process {} has received worker merging method ".format(rank) )

        for sentence in sentences:
            # Split the sentence into words
            words = sentence.split(" ")
```

```
            # Remove the first and last word.

            # Count the frequencies of the bigrams and unigrams in the sentence
            for i in range(len(words)-1 ):
                bigram = (words[i], words[i+1])
                bigram_counts[bigram] = bigram_counts.get(bigram, 0) + 1

                unigram = words[i]
                unigram_counts[unigram] = unigram_counts.get(unigram, 0) + 1




        if rank > 1:
            prev_bigram_counts = comm.recv(source=rank-1)
            prev_unigram_counts = comm.recv(source=rank-1)
            # Merge the data from the previous worker with our own data
            for bigram, count in prev_bigram_counts.items():
                if bigram not in bigram_counts:
                   bigram_counts[bigram] = 0
                bigram_counts[bigram] += count

            for unigram, count in prev_unigram_counts.items():
                if unigram not in unigram_counts:
                    unigram_counts[unigram] = 0
                unigram_counts[unigram] += count

        # Pass our data to the next worker (if any)
        if rank < num_procs -1:

            comm.send(bigram_counts, dest=rank+1)
            comm.send(unigram_counts, dest=rank+1)

        # If we are the last worker, the final data is in our bigram_counts and unigram_counts
dictionaries
        if rank == num_procs-1:
        # Use the merged data to calculate the conditional probabilities of the bigrams in the
test data
            comm.send(bigram_counts, dest=0)
            comm.send(unigram_counts, dest=0)
```

```
if merge_method=="MASTER" :
    if rank == 0:

    # Receive the bigram data from the worker processes
        for i in range(1, num_procs):
            received_data = comm.recv(source=i)

            # Merge the received data with the existing data
            for key, value in received_data.items():
                bigram_data[key] = bigram_data.get(key, 0) + value


        # Read the test file and split it into sentences
        with open(test_file, "r") as f:
            test_sentences = f.read().split("\n")

        # Compute the number of sentences
        number_of_sentences = len(test_sentences)


        for test_sentence in test_sentences:
            # Split the sentence into words
            words = test_sentence.split(" ")

            freq_of_first_word =  bigram_data.get(words[0]) or 0

            freq_of_both = bigram_data.get( (words[0], words[1]) ) or 0

            if freq_of_first_word > 0 :
                probs[test_sentence] =  freq_of_both / freq_of_first_word
            else:
                probs[test_sentence] = 0

            print ("conditional probability of {} is {}  " .format(test_sentence ,
probs[test_sentence]  )  )




# If merging method WORKERS is selected, conditional probabilities are calculated accordingly.
elif merge_method=="WORKERS" :
```

```
    if rank == 0:
        bigram_counts = comm.recv(source=num_procs-1)
        unigram_counts = comm.recv(source=num_procs-1)


         # Read the test file(test.txt) and split it into sentences
        with open(test_file, "r") as f:
            test_sentences = f.read().split("\n")


        for test_sentence in test_sentences:
            # Split the sentence into words
            words = test_sentence.split(" ")


            # Count the frequencies of the first word in the unigra_counts dictionary
            freq_of_first_word =  unigram_counts.get(words[0]) or 0

            # Count the frequencies of the bigram  in the bigram_data dictionary
            freq_of_both = bigram_counts.get( (words[0], words[1]) ) or 0


            if freq_of_first_word > 0 :
                probs[test_sentence] =  freq_of_both / freq_of_first_word
            else:
                probs[test_sentence] = 0

            #Prints the conditional probability of test sentence
            print ("(WORKER METHOD) Conditional probability of {} is {}  "
.format(test_sentence , probs[test_sentence]  )  )

# If the merging method is written incorrectly, this is handled.
else :

    if rank == 0:
        print("Choose a valid merge method: MASTER or WORKERS")
```