Cmpe300: Analysis of Algorithms
MPI Programming Project: Documentation

Course ID: CMPE300

Course Title: Analysis of Algorithms

Students: Deniz Ünal, Alp Eren İnceoğlu

Student IDs: 2019400234, 2019400063

Project Title: MPI Programming Project

Submission Date: 20.12.2022

Alp Eren İnceoğlu– 2019400063 20.12.2022
Deniz Ünal – 2019400234

Cmpe 300 Assignment 2: MPI Programming Project

Introduction:

In this project, we examined the n-gram system and created a python code which is capable of determining the frequency of all monograms and bigrams, which are single words and word couples which came consecutively, that were in the given huge input file and print the conditional probabilities of the required bigrams.

Program Interface and Execution:

While running this code user must use the correct formatting in the project call line and give parameter values accordingly. “mpi4py” must be downloaded and ready for use in the computer, as well as python. While running, user must run the program with a specified n, which will decide on the number of parallel nodes. If the specified n is larger than the core count of the system “--oversubscribe” argument also should be given. Other parameters are “--input_file”, “--merge_method” and “--test_file”, which all must be specified after the name of the python file in the call line. Input_file and test_file parameters must refer to the location of those files reachable from the current directory and the merge_method should either be ‘WORKERS’ or ‘MASTER’, which will change the way the code computes the data. A typical execution line should be as given:

mpiexec -n 12 --oversubscribe python DenizUnal.py --input_file data/sample_gnu.txt --merge_method WORKERS --test_file data/test_gnu.txt

Afterwards, the code must print the intended output to the command prompt and stop automatically.

Alp Eren İnceoğlu– 2019400063 20.12.2022
Deniz Ünal – 2019400234

Input and Output:

Other than the parameters as specified above, the input and test files must be formatted accordingly and have the locations given in the parameters. As for the formatting, input file must contain one sentence per each line and start and finish with “<s> “ and “ </s>” respectively. Test file however must contain only two word per line, which will indicate the intended 2-grams to print conditional probabilities of.

Program Structure:

Master:

Input: The code imports and uses “argparse” library in order to compute the parameters given in the command line, because of the fact that inputs weren’t given in order but rather by the name of the inputs, so using another system would require creating multiple possibilites for each ordering of the parameters. Using the ArgumentParser() method provided by the mentioned library, the code takes the parameters and stores them as args.input_file, args.test_file and args.merge_method. Afterwards, given files are read and stored; args.input_file as the list input_list and args.test_file as the string test_file. The reason that the input files are read in the master node and not send by their location names is that reading the file in each child would use unnecessary power, since it is common between all children nodes. Also, the parameters are also acquired here in order to be passed to the children too as it also would be an unnecessarily repeated action if done otherwise. For the input_file being a list, since each line will be examined on it’s own at one point, separating these while reading using the ‘readline()’ function in python rather than reading the thing as a whole and separating it afterwards.

```
parser=argparse.ArgumentParser()
parser.add_argument('--input_file')
parser.add_argument('--merge_method')
parser.add_argument('--test_file')
args=parser.parse_args()
size = comm.Get_size()

# Input file is read line by line and put into an array.
input_file = open(args.input_file)
line = input_file.readline()
input_list = []
while line:
    input_list.append(line)
    line = input_file.readline()
```

Figure 1: Reading arguments and sample file

Send: Since in the description it was strictly defined that the input must be separated as evenly as possible to the children nodes and python doesn't contain such a division method for integers, the code uses a simple algorithm to decide how many lines each node must receive. For this, the total number of lines are divided by the number of the child nodes and quotient and remainder are stored in the variables 'share' and 'extra' respectively. Then a for loop is opened to send the test_file, merge_method, number of total nodes and the relevant lines of the input_file as data to the relevant nodes in order. While deciding to the lines to be sent, if 'extra' is greater than 0, 'share'+1 lines are sent and 'extra' is decreased by 1. Also, data is sent to the nodes in the opposite order where the biggest child is sent first, since data is expected from the children in the same order for reasons explained below.

```
for i in range(size-1, 0, -1):
    # If there is an extra gives one more sentence to each worker
    # and decreases the extra sentence count.
    if extra > 0:
        end += 1
        extra -= 1
    # part is the array of sentences that the worker will get.
    part = input_list[start:end]
    # In the next itearation start of the part will be end of current
    # (array partitions doesn't involve the last index).
    start = end
    end += share
    # Each worker will get corresponding sentences, merge argument
    # and number of workers as its data.
    data={'input':part, 'size':size, 'merge_method':args.merge_method}
    comm.send(data, dest=i, tag=(10+i))
```

Figure 2: Sending data from master to workers

Receive: While merge method would normally change this part greatly and a big if would be required to decide which of the paths should be taken while receiving the n-grams, in this code we chose a much more elegant method. Instead of choosing between receiving from all children nodes or just the final node, we instead receive from the last node first and either stop receiving or continue the same loop according to the merge_method. Using the bi_count and mono_count dictionaries opened above the loop, we receive data from the appropriate children and combine the new dictionaries with the ones mentioned above in each loop. We use setdefault() in order to set the value of given keys either as just the new values or the sum of new value and the old value.

```
for i in range(size-1, 0, -1):
    # Gets the data from each worker and sums them to get the total count.
    recieved_data = comm.recv(source=i, tag=(20+i))
    recieved_bi = recieved_data['bigram']
    recieved_mono = recieved_data['monogram']
    for x in recieved_bi.keys():
        total_bi_count += recieved_bi[x]
        # setdefault is used to start the count from 0 if it is the first occurence of a bigram.
        bi_count[x] = bi_count.setdefault(x, 0) + recieved_bi[x]
    for x in recieved_mono.keys():
        total_mono_count += recieved_mono[x]
        mono_count[x] = mono_count.setdefault(x, 0) + recieved_mono[x]
    # If the merge type is workers master should only get data from the last worker.
    if args.merge_method == 'WORKERS': break
```

Figure 3: Master merging the data returned from workers

Print: Now that we have the number of all 1-grams and 2-grams, all that is left is to do is to take the value of the intended n-grams and print the division of those.

-

```
for test in test_list:
    # Skips if there is no bigram in an test file line.
    if test == '': continue
    # If are occurrences in the sample, key would be in dictionary so gets the information from there.
    if test in bi_count.keys():
        union_count = bi_count[test]
        single_count = mono_count[test.split(' ')[0]]
        print(test, union_count / single_count)
    # If there is no occurrence in the sample, key would not be in dictionary so prints 0 manually.
    else:
        print(test, 0, '(no occurrence in sample)')
```

Figure 4: Printing the probabilities

Children:

Input: The related data is received from the master node, and the length of the input file is used to determine how many lines this node is responsible of, which is then printed with the rank of the node, as requested in the description. Afterwards, the dictionaries named bi_count and mono_count are created. In a for loop for the input, each line is divided to it's words and the words are registered to the mono_count and their combination with the following word registered to bi_count, except for the last word in sentence in which case nothing is added to bi_count. This way, a single for loop is sufficient for filling both 1-grams and 2-grams, with the help of an additional if clause for the final word.

```
data = comm.recv(source=0, tag=(10+rank))
print('rank:', rank, '    number of sentences:', len(data['input']))

# bi_count and mono_count holds bigrams, monograms and their corresponding counts.
bi_count = {}
mono_count = {}

# The array that holds the input sentences read word by word
# and the counts are incremented accordingly.
for line in data['input']:
    line_arr = line.split(' ')
    for i in range(len(line_arr)):
        monogram = line_arr[i]
        mono_count[monogram] = mono_count.setdefault(monogram, 0) + 1
        if i < len(line_arr) - 1:
            bigram = ' '.join([line_arr[i], line_arr[i+1]])
            bi_count[bigram] = bi_count.setdefault(bigram, 0) + 1
```

Figure 5: Worker recieving data and processing it

Master: If the merge_method was given as MASTER in the data, node simply sends the data to the master node, without any complications.

```
# If the merge method is master all workers directly send data to master.
if data['merge_method'] == "MASTER":
    data = {'bigram':bi_count, 'monogram':mono_count}
    comm.send(data, dest=0, tag=(20+rank))
```

Figure 6: In master method all the merging is done by master, workers only send their own data

Workers: If not however, the node must check if it should receive any other data, which it only doesn't if it's rank is 1. If not, the data is received and combined with the already computed data in the same way it was combined in the master node. Afterwards, node sends this combined data to the next node, or if it's the final node to the master node.

```
elif data['merge_method'] == "WORKERS":
    # If rank is 1 there is no previous data.
    if rank > 1:
        recieved_data = comm.recv(source=(rank-1), tag=(14+rank))
        recieved_bi = recieved_data['bigram']
        recieved_mono = recieved_data['monogram']
        for x in recieved_bi.keys():
            bi_count[x] = bi_count.setdefault(x, 0) + recieved_bi[x]
        for x in recieved_mono.keys():
            mono_count[x] = mono_count.setdefault(x, 0) + recieved_mono[x]

    size = data['size']
    data={'bigram':bi_count, 'monogram':mono_count}

    # Sends it to next worker.
    if rank < size-1:
        comm.send(data, dest=(rank+1), tag=(15+rank))
    # Last worker sends the completed data to master.
    else:
        comm.send(data, dest=0, tag=(20+rank))
```

Figure 7: Workers merging the data and sending it

Alp Eren İnceoğlu– 2019400063 20.12.2022
Deniz Ünal – 2019400234

Examples:

For a given data folder containing the files “test_gnu”, which contains:

GNU /
GNU system

and “sample_gnu”, which contains:

I'd just like to interject for a moment.
What you're referring to as Linux, is in fact, GNU / Linux, or as I've recently taken to calling it, GNU plus Linux.
Linux is not an operating system unto itself, but rather another free component of a fully functioning GNU system made useful by the GNU corelibs, shell utilities and vital system components comprising a full OS as defined by POSIX.Many computer users run a modified version of the GNU system every day, without realizing it.
Through a peculiar turn of events, the version of GNU which is widely used today is often called "Linux", and many of its users are not aware that it is basically the GNU system, developed by the GNU Project.There really is a Linux, and these people are using it, but it is just a part of the system they use.
Linux is the kernel: the program in the system that allocates the machine's resources to the other programs that you run.
The kernel is an essential part of an operating system, but useless by itself; it can only function in the context of a complete operating system.
Linux is normally used in combination with the GNU operating system: the whole system is basically GNU with Linux added, or GNU / Linux.
All the so-called "Linux" distributions are really distributions of GNU / Linux

The code is run using the call line:

```
mpiexec -n 5 –oversubscribe python DenizUnal.py --input_file data/sample_gnu.txt --merge_method WORKERS --test_file data/test_gnu.txt
```

The output in the command prompt is:

```
rank: 4     number of sentences: 2
rank: 2     number of sentences: 2
rank: 3     number of sentences: 2
rank: 1     number of sentences: 2
GNU / 0.25
GNU system 0.16666666666666666
```

As it can be seen, the code runs exactly how it should be accoring to the initial description and the way it was explained above.

Improvements and Extensions:

Since this was a straightforward project with clear goals and all goals were reached, there aren't any possible improvements that could be done on the code. However, for a more complex code this one can be edited to detect different n-grams with different inputs and calculate more difficult conditional probabilities with little change. Also an option to ignore the punctutations could be added as an extension. For example in the example output GNU word occurs total of 12 times and in 2 of these occurrences followed by "system" and one is followed by "system,". In our case we didn't add any feature to ignore punctuations such as this but it could be implemented as an option.

Difficulties Encountered:

Downloading mpi4py was probably the biggest roadblock we faced, as the provided download method was utterly useless and long searches on the internet for download instructions hardly provided a possible way for linux. However for Windows it was even harder, and only many failed tries, applications and long download times finally resulted in managing to download the library using another application called 'anaconda'.
Other than that, trying to find a way to simplify the different merge methods system was kind of exhausting and was only reached after some thought was put onto the different aspects of reversing the node orders.

Alp Eren İnceoğlu– 2019400063 20.12.2022
Deniz Ünal – 2019400234

Conclusion:

While this was a relatively easy project, it was still a great introduction to parallel processes and a more interactive way of learning the n-grams and conditional probabilities. Our code in this project was focused on trying to be as efficient as possible, since the use of parallel processes automatically shows such an intend. As for the final product, the python program can compute the intended output in a short time without any errors, which we believe was a satisfying result.

Appendices:

Anaconda | The World's Most Popular Data Science Platform

```
#Student Name: Deniz Ünal
#Student Number: 2019400234
#Compile Status: Compiling
#Program Status: Working

from mpi4py import MPI
import argparse

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

if rank == 0:
    # To get the arguments with the corresponding flags argparse module is used.
    parser=argparse.ArgumentParser()
    parser.add_argument('--input_file')
    parser.add_argument('--merge_method')
    parser.add_argument('--test_file')
    args=parser.parse_args()
    size = comm.Get_size()

    # Input file is read line by line and put into an array.
    input_file = open(args.input_file)
    line = input_file.readline()
    input_list = []
    while line:
        input_list.append(line)
        line = input_file.readline()
    # share is the number of sentences each worker gets.
    share = len(input_list) // (size-1)
    # extra is there for the cases which sentence count doesn't dvisible by worker count.
    # In such a case extra is distributed to some workers one by one.
    extra = len(input_list) % (size-1)
    start = 0
    # end is the index which current partition of the array will end.
    end = share
    input_file.close()
```

```
    # Test file is read and split into an array which all indexes corresponds to a bigram.
    test_file = open(args.test_file)
    test_str = test_file.read()
    test_list = test_str.split('\n')

    for i in range(size-1, 0, -1):
        # If there is an extra gives one more sentence to each worker
        # and decreases the extra sentence count.
        if extra > 0:
            end += 1
            extra -= 1
        # part is the array of sentences that the worker will get.
        part = input_list[start:end]
        # In the next itearation start of the part will be end of current
        # (array partitions doesn't involve the last index).
        start = end
        end += share
        # Each worker will get corresponding sentences, merge argument
        # and number of workers as its data.
        data={'input':part, 'size':size, 'merge_method':args.merge_method}
        comm.send(data, dest=i, tag=(10+i))

    # bi_count and mono_count holds bigrams, monograms and their corresponding
counts.
    bi_count = {}
    mono_count = {}
    total_bi_count = 0
    total_mono_count = 0

    for i in range(size-1, 0, -1):
        # Gets the data from each worker and sums them to get the total count.
        recieved_data = comm.recv(source=i, tag=(20+i))
        recieved_bi = recieved_data['bigram']
        recieved_mono = recieved_data['monogram']
        for x in recieved_bi.keys():
            total_bi_count += recieved_bi[x]
            # setdefault is used to start the count from 0 if it is the first occurence of a bigram.
            bi_count[x] = bi_count.setdefault(x, 0) + recieved_bi[x]
        for x in recieved_mono.keys():
            total_mono_count += recieved_mono[x]
            mono_count[x] = mono_count.setdefault(x, 0) + recieved_mono[x]
        # If the merge type is workers master should only get data from the last worker.
        if args.merge_method == 'WORKERS': break

    # Gets each bigram from the array that holds bigrams from the test file.
    # Gets the count for both of them and divides it to the count for the
    # first word to get the probability of the bigram and prints it.
    for test in test_list:
        # Skips if there is no bigram in an test file line.
        if test == '': continue
        # If are occurrences in the sample, key would be in dictionary so gets the
information from there.
        if test in bi_count.keys():
            union_count = bi_count[test]
            single_count = mono_count[test.split(' ')[0]]
            print(test, union_count / single_count)
```

```
        # If there is no occurrence in the sample, key would not be in dictionary so prints 0
manually.
        else:
            print(test, 0, '(no occurrence in sample)')

else:
    data = comm.recv(source=0, tag=(10+rank))
    print('rank:', rank, '    number of sentences:', len(data['input']))

    # bi_count and mono_count holds bigrams, monograms and their corresponding
counts.
    bi_count = {}
    mono_count = {}

    # The array that holds the input sentences read word by word
    # and the counts are incremented accordingly.
    for line in data['input']:
        line_arr = line.split(' ')
        for i in range(len(line_arr)):
            monogram = line_arr[i]
            mono_count[monogram] = mono_count.setdefault(monogram, 0) + 1
            if i < len(line_arr) - 1:
                bigram = ' '.join([line_arr[i], line_arr[i+1]])
                bi_count[bigram] = bi_count.setdefault(bigram, 0) + 1

    # If the merge method is master all workers directly send data to master.
    if data['merge_method'] == "MASTER":
        data = {'bigram':bi_count, 'monogram':mono_count}
        comm.send(data, dest=0, tag=(20+rank))
    # If the merge method is workers each worker gets data from
    # previous worker merges it with its own data and sends it to next worker.
    elif data['merge_method'] == "WORKERS":
        # If rank is 1 there is no previous data.
        if rank > 1:
            recieved_data = comm.recv(source=(rank-1), tag=(14+rank))
            recieved_bi = recieved_data['bigram']
            recieved_mono = recieved_data['monogram']
            for x in recieved_bi.keys():
                bi_count[x] = bi_count.setdefault(x, 0) + recieved_bi[x]
            for x in recieved_mono.keys():
                mono_count[x] = mono_count.setdefault(x, 0) + recieved_mono[x]

        size = data['size']
        data={'bigram':bi_count, 'monogram':mono_count}

        # Sends it to next worker.
        if rank < size-1:
            comm.send(data, dest=(rank+1), tag=(15+rank))
        # Last worker sends the completed data to master.
        else:
            comm.send(data, dest=0, tag=(20+rank))
    # There is no other valid merge method so throws error.
    else:
        print("Invalid merge method, terminating the program.")
        exit()
```