Bogazici University Department of Computer Engineering

CMPE 300: Analysis of Algorithms

MapReduce

Application Programming Project Report

Student: Buse Ece Guldiken
Number: 2015400087

Submitted to: Mehmet Kose
Date: 18.12.2017

1) Introduction

This project aims to count the word occurrences in a file using MPI Parallel Programming. The problem is solved using Open MPI environment in C++ language. Project follows an algorithm similar to MapReduce, explained with a figure below.[1]

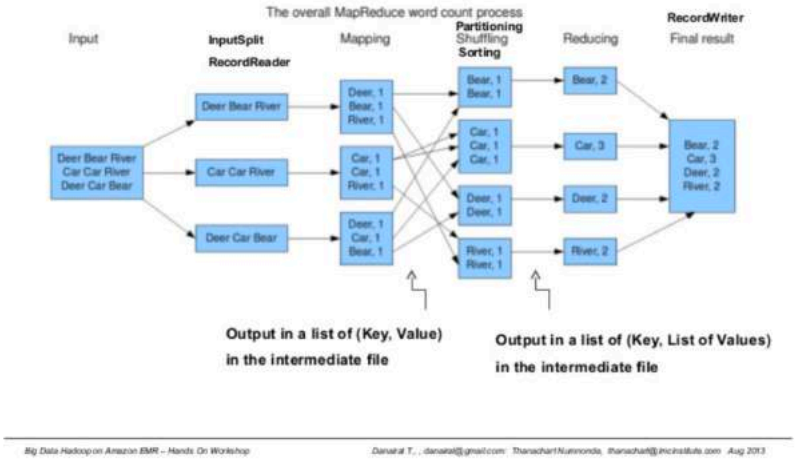

Figure 1. MapReduce algorithm schema

There are 7 main steps in the program flow:

1. Master process reads the input from speech_tokenized.txt
2. Master process splits input to slave processors
3. Slaves change occurrence of each word to 1 and send it back to the master
4. Master process splits the word-count to slave processors
5. Slave processors sort the input with respect to alphabetical order and send it back to the master
6. Master process gathers all sorted inputs and sorts them
7. Master process reduces the sorted list in alphabetical order

There is one master process and (size-1) number of slave processors, given as an argument after -np.

Assumptions and requirements regarding the project:

- Input file will given tokenized, there will be one word at each line.
- The number of processors given will be less than or equal to number of tasks.
- Input and output handled by master processor.
- In case of empty file, program flow will be interrupted after input reading.
- Number of words given to each processor may vary depending on input size and number of processors.

2) Program Interface

The program is written in Mac OS environment with Open Mpi using C++ language.

Requirements to run the project:

- Open mpi (Installing tutorial at Appendix)
- Working C++ Compiler (gcc)

After requirements are satisfied user can compile the program with command:

mpic++ <source_code_name>.cpp -o <program_name>

Ex: mpic++ main.cpp -o mainProg

And can run the program with command:

mpirun -np <number_of_processors> ./<program_name>

Ex: mpirun -np 2 ./mainProg

The program automatically terminates when it finishes writing number of occurrences to terminal window.

3) Program Execution

Example of an ordinary input with 4 processors

Input file named speech_tokenized.txt is written. [6]

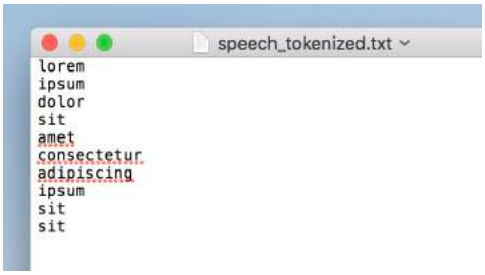

Figure 6- Input file to be used

Program code file is named inputdene.cpp and it is compiled and runned by commands:

```
mpic++ inputdene.cpp -o inputdene
mpirun -np 4 ./inputdene
```

Master process (process 0) reads the input file and writes the words with their counts 0. [7]

```
Buse-MacBook-Pro:Desktop buseece$ mpic++ inputdene.cpp -o inputdene
Buse-MacBook-Pro:Desktop buseece$ mpirun -np 4 ./inputdene
lorem 0
ipsum 0
dolor 0
sit 0
amet 0
consectetur 0
adipiscing 0
ipsum 0
sit 0
sit 0
```

Figure 7- Master reads the input file

Master process splits the data almost equally among other processors. [8]

```
1 will receive 4 words starting from index 0
2 will receive 3 words starting from index 4
3 will receive 3 words starting from index 7
```

Figure 8 - Processor counts

Each process counts the words at their local buffer.

Master process calls gathering function and calls split function again to have inputs sorted.[9]

```
1 will sort
lorem 1
ipsum 1
dolor 1
sit 1
```

Figure 9 - Example data at processor 1

Each process sorts local splitted data.[10]

```
1 sorted:
dolor 1
ipsum 1
lorem 1
sit 1
```

Figure 10- Example sorted data at processor 1

After local sorting is done, master calls gathering function and resorts the global array.

Master reduces counts using map.[11]

```
adipiscing :: 1
amet :: 1
consectetur :: 1
dolor :: 1
ipsum :: 2
lorem :: 1
sit :: 3
Buse-MacBook-Pro:Desktop buseece$
```

Figure 11 - Reduced list of words and counts

4) Input and Output

Input File:

To be able to run the program, input file should consist of tokenized words and should be named *speech_tokenized.txt*.[2]

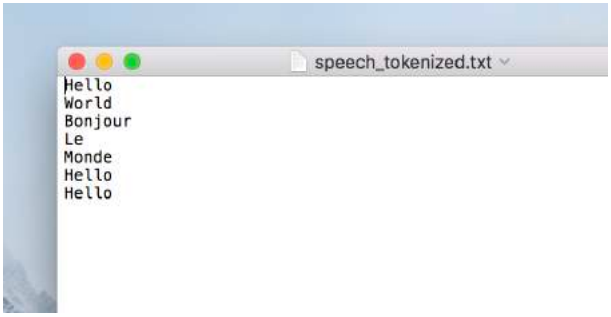

Figure 2 - example input file

Input file should be at the same directory as the source code. If not, the address of the input file should be specified/changed at the code. *See line 52 at the code (ifstream inFile ("./speech_tokenized.txt")).*

Output:

After successfully compile & run output of the program will be written at terminal window. Output will contain all words and their number of occurrences alphabetically sorted, with the form: <word1> :: <count1>.[3]

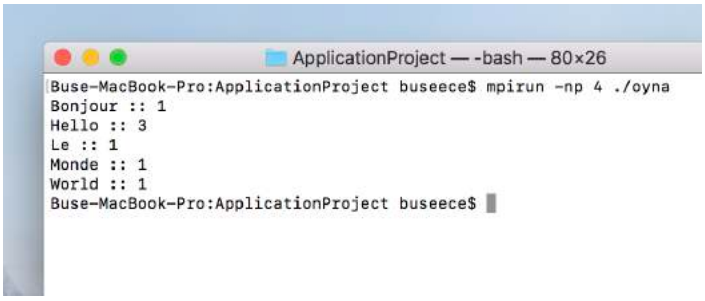

Figure 3 - example output

5) Program Structure

The code is under a single file named “main.cpp” consisting of 190 lines with comments.

As mentioned at the introduction section, the program roughly consists of 7 parts.

1. Master process reads the input from speech_tokenized.txt
2. Master process splits input to slave processors
3. Slaves change occurrence of each word to 1 and send it back to the master
4. Master process splits the word-count to slave processors
5. Slave processors sort the input with respect to alphabetical order and send it back to the master
6. Master process gathers all sorted inputs and sorts them
7. Master process reduces the sorted list in alphabetical order

1.Declarations and Initializations

1.1.Declared fields and interpretations:

- int size
  number of processors
- int rank
  overall rank of a processor
- int inputLength
  length of the tokenized input file (# of words)
- int *counts
  number of words that will be sent to each processor
- int *displays
  starting index of words that will be sent to each processor
- int elements_per_proc
  number of words per processor without remainder
- int max_elements_per_proc
  number of words per processor with remainder (maximum number of inputs among all processors)
- int rem
  remainder that will be distributed partially equal among processors

- WordCount receiveBuffer[max_elements_per_proc]
  processors’ local receive buffer of size max_elements_per_proc
- vector<string> inputs
  input file is read and words are kept in this vector

1.2.MPI Initialization

- MPI_Init(&argc, &argv):
  initializes MPI with given arguments
- MPI_Comm_size(MPI_COMM_WORLD, &size):
  initializes number of processors in the environment
- MPI_Comm_rank(MPI_COMM_WORLD, &rank):
  initializes rank of the processor in the environment

1.3.Structs:

- Struct WordCount:
  It is implemented in order to store the string as a char array and its number of occurrences.
- Struct WordCount globalWords[inputLength]:
  Global data is stored under globalWords. It contains all words and their occurrences.

1.4.Scatterv helper structures:

- counts = new int[size]:
  It stores number of words that will be sent to each processor at respective indexes.
- displays = new int[size]:
  It stores displacement index of words that will be sent to each processor at respective indexes.

2.Getting input and splitting

2.1.Getting input from the file

Only master process interacts with the input file, it reads each line and stores each word in vector<string> inputs. Then it gets input file length via inputs.size(), and it decides how many words will take each processor.

2.2.Splitting the input

Master processor changes *counts[size]* and *displays[size]* arrays with respect to input size and number of processors.

Example usage of the arrays:

- counts[0] = 4 //meaning processor 0 will receive 4 data
- counts[1] = 3 //meaning processor 1 will receive 3 data

- displays[0] = 0 //meaning data that will receive will start from index 0 of the global array
- displays[1] = 4 // meaning data that will receive will start from index 4 of the global array

Processor 0	Processor 0	Processor 0	Processor 0	Processor 1	Processor 1	Processor 1
index 0				index 4		

Scatterv function is used in order to accomplish the function. (See manual for Scatterv)

Example call with Scatterv:

```
MPI_Scatterv(globalWords, counts, displays, Particletype, receiveBuffer,counts[rank],
Particletype, 0, MPI_COMM_WORLD);
```

3.Gathering input from slaves

After receiving the input slaves change occurrence of each word to 1 and master processor gathers this updated version by using gathering function.

Since input sizes are different *Gatherv* function is used in order to gather partitioned input from slaves.

Example call with Gatherv:

```
MPI_Gatherv(receiveBuffer, counts[rank],  Particletype, globalWords, counts,
displays, Particletype,0, MPI_COMM_WORLD);
```

4.Splitting data to be sorted

4.1.Slaves sort their partitions

Master recalls Scatterv function to split array of words and their occurrences to slave processors. Processors calls qsort function with user defined compare method *struct_cmp_by_word(const void *a, const void *b).*

Example call to qsort function:

```
qsort( receiveBuffer, numberOfWords, sizeof *receiveBuffer, struct_cmp_by_word );
```

qsort function sorts *receiveBuffer* elements (WordCount instances) with respect to alphabetical order of char array field word. The compare method is overwritten by struct_cmp_by_word.

```
int struct_cmp_by_word(const void *a, const void *b) {
    struct WordCount *ia = (struct WordCount *)a; //reference & dereference
    struct WordCount *ib = (struct WordCount *)b;
    return strcmp(ia->word, ib->word);//compares two char arrays
}
```

Compare method compares two char arrays and returns 1 if word1 should come before, as it is in strcmp(arg1,arg2) method.

4.2. Master sorts the entire array

After slaves finish sorting their local partitions, master calls again Gatherv function to gather scattered data.

It calls qsort function to sort the entire data. (See details under 4.1.Slaves sort their partition).

5. Master reduces the number

After sorting the list master creates a Map structure to hold word and its count value.

```
map<string,int> globalMap; //Create a map
```

Example mapping pairs:
globalMap: {<Hello,3>, <World,2> ..}

It searches among globalData list and creates a new key value if the word is not added yet to the map, otherwise it just increments the count of occurrences in respective amount.

6. Master writes words and their occurrences to terminal window

Using iterator over globalMap (map of word and count pairs), master processor prints the key and values to terminal window following the form: <word> :: <count>.

(See Figure 3 for example output.)

6) Examples

One example program execution is given at Program Execution section.

All below examples are run with 4 processor command.

Example 1 Input:

speech_tokenized.txt

Hello

Bonjour

Le

Monde

Hello

Hello

Example 1 Output:

Bonjour :: 1

Hello :: 3

Le :: 1

Monde :: 1

World :: 1

Example 2 Input:

speech_tokenized.txt (Empty)

Example 2 Output:

empty file

Example 3 Input:

speech_tokenized.txt

a

a

a

Example 3 Output:

a :: 3

7) Improvements and Extensions

The program might be written in more efficient way with respect to time and space complexity. But since the aim was to understand the parallel programming approach, major focus was on learning parallel programming.

Also in this version program does not run if number of processors are more than number of parallelized tasks. It can be improved such that a processor can stay idle through the run.

8) Difficulties Encountered

During the project major difficulty was the lack of examples and sources on the Internet. The tutorial and the examples provided at the websites were too basic that understanding the MPI functions required numerous trials and failures.

The second biggest difficulty was because of programming languages that might be used with MPI interface. Even C++ language feels closer, the coder should have broad knowledge to use pointers and structs. It was really time consuming to understand where to declare and how to initialize the structures.

It was hard to debug to errorneous code when it fails due to its parrallel struct but the harder thing was to decide if a statement will be seen by all processors or not. I tried hard to understand Scatterv and Gatherv functions, first I have implemented it such that I was broadcasting counts and displays arrays. My code was failing time to time, then debugging took me at least 5 hours because it was impossible to understand where is the problem(more than hours that I wrote it). Then I understood that problem was to broadcasting these arrays. I declared them in common area to all processors and with trials I found the correct way.

9) Conclusion

This project was a good practice to understand and learn the basic functions regarding parallel programming approach. Difficulties of the project were to understand Gatherv and Scatterv functions and to create Struct of the correct form. Once the concepts are understood, actually it is not that complex project. Indeed it was a hard project to start but also I think it is beneficial to learn the complicated functions.

It was a good opportunity to have hands on experience to familiarize with the MPI programming interface, but one needs to practice more to write better programs.

10)Appendices

8.1.Installing Open MPI under Linux and Mac OS X

1. Download latest stable release of open mpi from

   http://www.open-mpi.org/software/ompi/v1.4/downloads/openmpi-1.4.4.tar.gz

2. Type these commands to build Open MPI with default settings (if want to change defaults, find and read the Open MPI installation guide)

   tar -xvf openmpi-1.4.4.tar.gz ./configure
   make all install

3. Now, go to the directory openmpi-1.4.4/examples to compile the examples

   Type ”make”
   Three examples will be build: hello c.c, ring c.c and connectivity c.c

   Run hello c by typing ”mpiexec -n 2 ./hello c”

If you get the error ”mpicc: error while loading shared libraries: libopen-pal.so.0: cannot open shared object file: No such file or directory”, run the command ”ldconfig” (as root) to update the shared library bindings (for more info: http://linux.die.net/man/8/ldconfig)

How to compile your code with Open MPI:

mpicc -g your_code.c -o your_program

How to run your code with Open MPI:

mpiexec -n NUM_PROCESSORS ./your_program

8.2.Code

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <iterator>

using namespace std;

//Struct WordCount that stores a word and number of occurrences
struct WordCount
{
    int count; //initially 0
    char word[100]; //word is stored as char array
};


// qsort struct comparison function stated as its tutorial
int struct_cmp_by_word(const void *a, const void *b) {
    struct WordCount *ia = (struct WordCount *)a; //reference & dereference
    struct WordCount *ib = (struct WordCount *)b;
    return strcmp(ia->word, ib->word);//compares two char arrays
}

int main(int argc, char *argv[])
{
    int size,rank; //number of processors, overall rank of a processor
    int inputLength; //length of tokenized input file

    int *counts=NULL; //stores number of words to-be-sent to a processor (Scatterv)
    int *displays=NULL; //stores starting index of words to-be-sent to a processor
(Scatterv)

    int elements_per_proc; // input length / number of processors
    int max_elements_per_proc; // (input length / number of processors) +1 if
remainder>0
    int rem; //input length % number of processors

    int sum=0; //not used

    //WordCount receiveBuffer[1000]; //processors' local receive buffer

    //MPI Initialization
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size); //initialize size
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //initialize rank

    vector<string> inputs; //to get unknown number of input from the file

    //Master process gets input from the file
    if (rank == 0) {
        ifstream inFile("./speech_tokenized.txt"); //assumed the file name
        //string inputFile = argv[1];
        //ifstream inFile(inputFile.c_str());
        string input="";
        if(inFile.is_open()){ //if file is open
            while(inFile >> input){ //get new token
                inputs.push_back(input); //put it in inputs vector
            }
        }
        inputLength = inputs.size(); //input length is number of words
        if(inputLength == 0){ //if zero, exit
            cout<<"empty file "<<endl;
            return 1;
```

```
        }

    }

    //Declare struct to store words and their counts
    struct WordCount globalWords[inputLength];
    MPI_Status status;
    MPI_Datatype Particletype; //its type
    MPI_Datatype type[2] = { MPI_INT, MPI_CHAR }; //int and char[] thus MPI_INT,
MPI_CHAR
    int blocklen[2] = { 1, 100 }; // will store 1 int and 100 chars
    MPI_Aint intex;
    MPI_Type_extent(MPI_INT, &intex); //size of int
    MPI_Aint disp[2]; //starting indexes
    disp[0] = (MPI_Aint) 0; //starts from 0
    disp[1] = intex; // starts after first field

    MPI_Type_create_struct(2, blocklen, disp, type, &Particletype); //create the struct
    MPI_Type_commit(&Particletype); //and commit to see

    counts = new int[size]; //number of words
    displays = new int[size]; //where it starts in the globalWords array

    if (rank == 0) {
        string try0="";
        int try1=0;
        //initialize array of struct wordCount
        for(int i=0; i<inputLength; i++){
            globalWords[i].count=0; //at the beginning their count is unknown
            strcpy(globalWords[i].word, inputs.at(i).c_str()); //take the char array
        }
    }

    //Broadcast the inputLength, # of elements per proc, max # of elements per proc to
slaves
    MPI_Bcast(&inputLength, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    //Broadcast counts array to slaves
    //cout<<inputLength<<endl;
    elements_per_proc = inputLength / (size-1); // divide words to procs without
processor 0
    rem = inputLength % (size-1); //remainder that will be distributed to words

    //calculate how many words that each process will take
    counts[0] =0; //0 processor will not calculate
    displays[0]=0; //0 processor will not calculate
    max_elements_per_proc = elements_per_proc;
    for(int i=1; i< size; i++){ // for each slave processor
        counts[i] = elements_per_proc; //if there was no remainder
        if(rem > 0){ //if there is a remainder give 1 to this processor
            counts[i]++;
            max_elements_per_proc = counts[i]; //this will be maximum number of words by
any processor
            rem--;
        }
        displays[i]= sum; // stores index of start for this processor
        sum += counts[i];
    }

    WordCount receiveBuffer[max_elements_per_proc]; //processors' local receive buffer

    //if there will be idle processes exits
    for(int i=1;i<size; i++){
        if(counts[i]==0){ //that means processor with no input
            cout<<"more processes than tasks give less processes "<<endl;
            return 1; //return back
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
```

```
    //Split input data to slaves with respect to counts and displays
    MPI_Scatterv(&globalWords, counts, displays, Particletype, &receiveBuffer,
counts[rank], Particletype, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    for(int i=0;i<counts[rank]; i++){
        receiveBuffer[i].count = 1; //change word counts to 1
    }

    //Gather all inputs from slaves
    MPI_Gatherv(receiveBuffer, counts[rank],  Particletype, globalWords, counts,
displays, Particletype,0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

    //Scatter again the array, with word counts 1
    MPI_Scatterv(globalWords, counts, displays, Particletype, receiveBuffer,
counts[rank], Particletype, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    //For slaves
    if(rank!=0){
        int numberOfWords = counts[rank]; //number of words it received
        qsort( receiveBuffer, numberOfWords, sizeof *receiveBuffer, struct_cmp_by_word
); //sort the structs in alphabetical word order
    }

    MPI_Barrier(MPI_COMM_WORLD); //Waiting for each processor to finish before gathering

    //Gather all inputs from slaves, from localWords to globalWords
    MPI_Gatherv(receiveBuffer, counts[rank],  Particletype, globalWords, counts,
displays, Particletype,0, MPI_COMM_WORLD);

    if(rank==0){
        //Sort gathered inputs at the master
        qsort( globalWords, inputLength, sizeof *globalWords, struct_cmp_by_word );
        //Reduce gathered and sorted input at master
        map<string,int> globalMap; //Create a map
        string _word="";
        int _count=0;
        for(int i=0; i<inputLength; i++){ //For each word
            string str(globalWords[i].word);
            _word = str; // take word
            _count = globalWords[i].count; //take its count
            int oldNum, newNum=0;
            if(globalMap.find(_word)!= globalMap.end()){ //if word is already added to
map, update its number of occurrence
                oldNum= globalMap.find(_word)->second; //get old count
                newNum = oldNum + _count;
                globalMap.find(_word)->second = newNum; //update the count
            }else{ // word is not found in the map, add it
                globalMap.insert(make_pair(_word, _count));
            }
        }
        std::map<std::string, int>::iterator it = globalMap.begin(); //create iterator
to move along the map
        while(it != globalMap.end()){ //while its not the end of the map
            std::cout<<it->first<<" :: "<<it->second<<std::endl; //write key :: value
            it++;
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```