Bogazici University

cmpe 300

Analysis of Algorithms

Map Reduce Programming Project

Author: Ahmet Ege MAHLEÇ
2016800045

Submitted to: Mehmet Köse

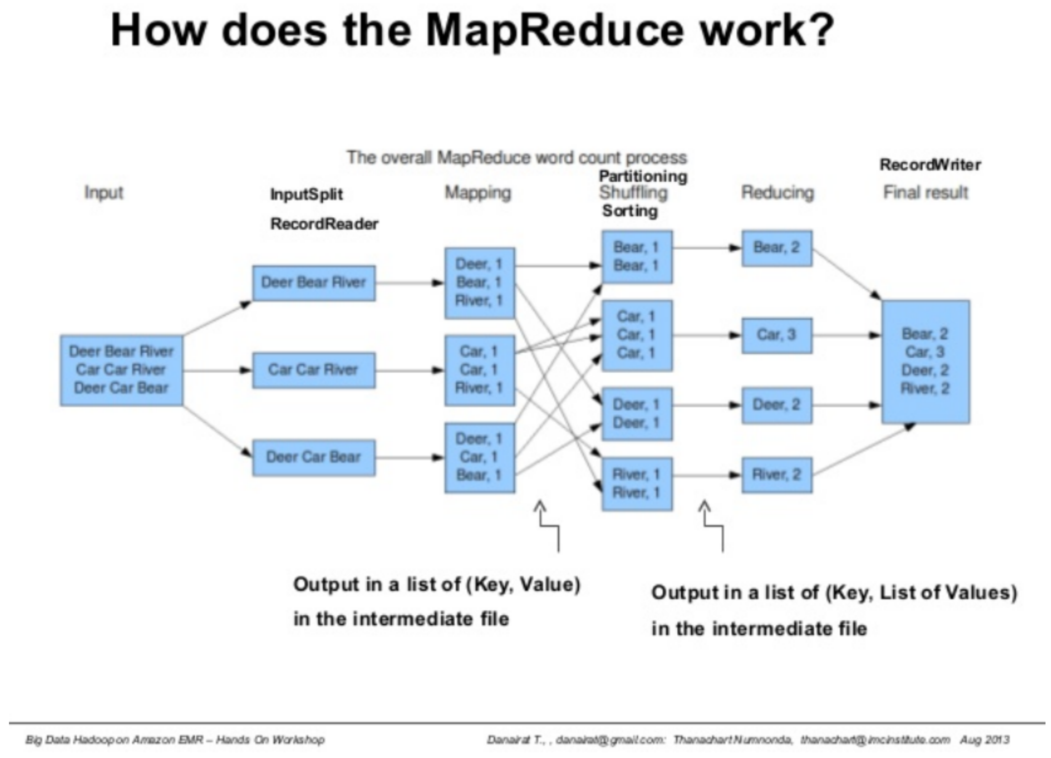
December 18, 2017

Contents

1 Introduction

This report is dedicated to explain the design of Map Reduce Algorithm to count the word occurrence in a file. Map Reduce Algorithm is basically explained in the Figure 1 [1] as it is given in the description of the project.

Figure 1: Map Reduce

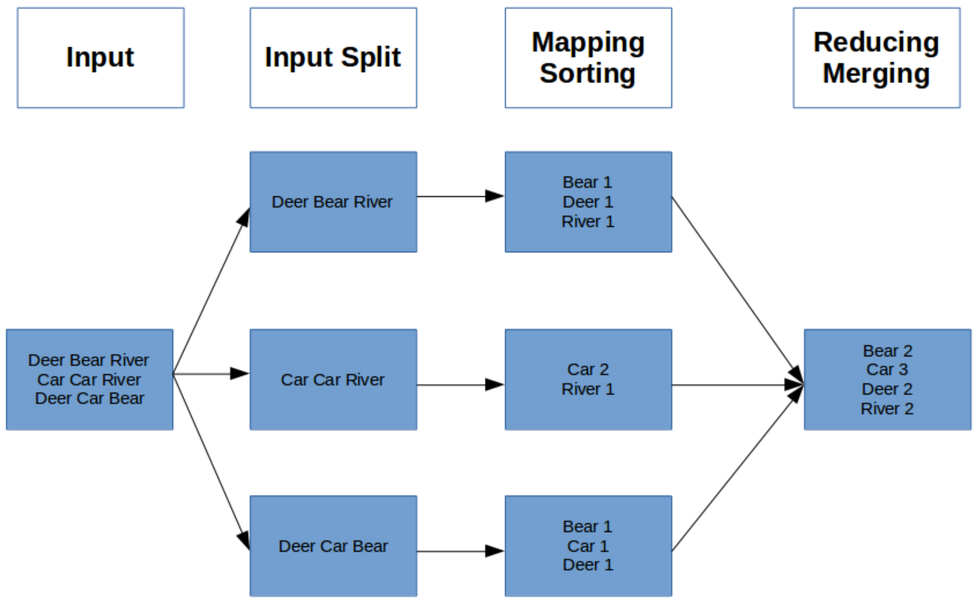


As it is shown in the Figure 1, root process will read the input file and split the file into n-1 partitions where n denotes number of processes including the root process. While splitting the file, it is important to note that the size of the partitions should be almost equal so that every process will have almost equal load. We are saying almost equal since while splitting file into partitions, we need to be careful not to split any word into two. Therefore, we need to check whether the brackets are onto white space.

After the file is divided into n-1 partitions, root process should send data which belongs to corresponding partition to the process whose rank is equal to partition's number. Then, every process will map the words occurred in that partition. In addition to that, every process will sort the words occurred in the corresponding partition in lexicographic order. After that, every process will send output data to root process so that root process merges (reduces) the output into one and print the result.

Since the implementation is different than the project description given, it is better the draw a new diagram to explain my implementation as it is shown in figure 2.

Figure 2: Customized Map Reduce



As it can be seen in the figures 1 and 2, the steps are not same for the methods. In the Customized Map Reduce Algorithm hasn't got shuffling stage. In the Classical Map Reduce Algorithm, after map stage is done, words are directed to corresponding processes by using hashing in the shuffling stage. In our Customized Map Reduce Algorithm, we are not using shuffling.

2 Assumptions and Constraints

In this section, assumptions and constraints will be given.

- Characters which is not defined in ASCII table will not be given as input since every character is stored in one byte.In other words, the input should be in English.
- According to Oxford dictionary, the longest word is 45-character long. Therefore, the buffer to store a word is reserved as 50-byte long. [2]
- The size of the input file should be less than the free space of the machine which hosts the root process.
- The size of the partition should be less than the free space of the machine which host corresponding process.
- Even there is enough free space in the machine, the size of the file cannot be more than 4GB since data size is stored in unsigned integer variable. (Assuming that integer variable is 4-byte long)
- The program can take multiple inputs. Every inputs will be evaluated one by one.
- If the input cannot be divided n partitions where n is the number of processes, less than n process will be used depend on the number of partitions.
- This report is prepared for Linux Users. All the programming explanation and execution commands are for Linux Users.

3 Program Interface

The program is written by using C++ language and Open MPI. Open MPI is an open source implementation that is developed and maintained by consortium of academic, research, and industry partners. In order to execute the program, Open MPI must be installed in user's system. For further information to install Open MPI, please check the links [3][4] given in the appendices.

After Open MPI is installed, user should enter the directory which source code exists and call the following command to compile the source codes.

```
$ mpic++ -o <program> main.cpp ProcessManager.cpp
   FileSplitter.cpp WordCounter.cpp
```

mpic++ is Open MPI C++ wrapper compiler. It is called as "wrapper" compiler since they do not actually compile or link applications themselves; they only add in command line flags and invoke the back-end compiler. Since they are using C++ compiler as back-end compiler, the arguments is same as C++ compiler. -o stands for the output executable name. User can choose any name as executable name. The other parameters are the source codes.

After the program is built, it can be executed by using the following command.

```
$ mpirun [-np X] [--hostfile <filename>] <program> <input>
```

This command will run X copy of ¡program¿ in user current run-time environment. All the copy of ¡program¿ will take input; however, only root process will open the input and print the output.

In case user would like to run more processes than number of processors, they can use –hostfile parameter by creating an host file. In the host file, user can choose the number of slots and user can assign the processes to any processor s/he wants. For more information, please visit the link.[6]

4 Program Execution

This section is divided into three parts, namely, How To Use, Input and Output.

4.1 How To Use

Launching the program is expressed in the section 3. After the program is launched, user will see the result in the terminal window and program will be terminated. No user interface was designed since the program is very simple. The program will take the input and print the result.

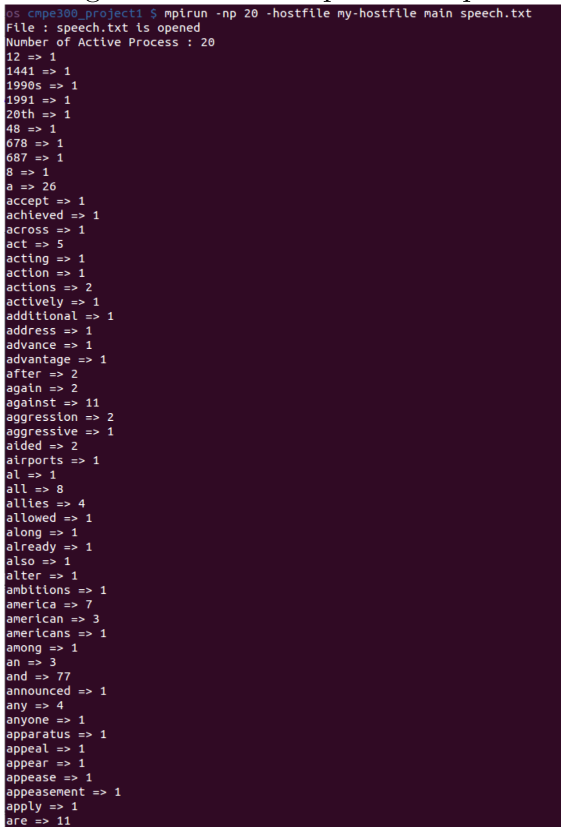
4.2 Input

The program can take any input whose type is Plain Text Document. PDF or Microsoft Word Document cannot be given to program as input. As long as Plain Text Document is given to program, it doesn't matter whether the input is tokenized or not. Input will be formed to tokenized format by the program even if it has already been tokenized.

4.3 Output

The program prints the result on terminal window and terminate the program. An example of output can be found in the Figure 3.

Figure 3: An Example of Output



```
os cmpe300_project1 $ mpirun -np 20 -hostfile my-hostfile main speech.txt
File : speech.txt is opened
Number of Active Process : 20
12 => 1
1441 => 1
1990s => 1
1991 => 1
20th => 1
48 => 1
678 => 1
687 => 1
8 => 1
a => 26
accept => 1
achieved => 1
across => 1
act => 5
acting => 1
action => 1
actions => 2
actively => 1
additional => 1
address => 1
advance => 1
advantage => 1
after => 2
again => 2
against => 11
aggression => 2
aggressive => 1
aided => 2
airports => 1
al => 1
all => 8
allies => 4
allowed => 1
along => 1
already => 1
also => 1
alter => 1
ambitions => 1
america => 7
american => 3
americans => 1
among => 1
an => 3
and => 77
announced => 1
any => 4
anyone => 1
apparatus => 1
appeal => 1
appear => 1
appease => 1
appeasement => 1
apply => 1
are => 11
```

As it can be seen in the Figure 3, the program prints the "Number of Active Process". This information will indicate how many process will be used to count word occurrences in input file. The number of active process will be less than or equal to number of process which is given as input argument via "**-np**". This topic will be explained in detail in the next section.

5 Program Structure

In this section, technical details of the program will be explained. This section will be presented in two parts; namely, Methods and Data Structures.

5.1 Methods

In this subsection, main method and static methods of classes will be explained.

5.1.1 Main Method

In the ”Main” method located in main.cpp file, the following items are managed.

- MPI environment is set up.
- Root process’ and slave processes’ code is separated so that they behaves differently.
- Root process opens the file and divide the input n-1 almost equal partitions where n is the number of processes so that root process send the partitions to slave processes.
- Root process will check the partitions whether the bracket is on white space or not. We need to be careful that we shouldn’t divide any word into two.
- In case input is too small to divide n-1 partitions where n is the number processes, number of active processes will be calculated.
- Root process will send ”nActiveProcessFlag” variable to delegate the slave process whether it is active or passive. If slave process is passive, it will do nothing. If slave process is active, it will receive an integer to learn the size of the data which will be sent by root process and after that it will receive the data.
- Slave processes will map and sort the partitions and send the results to root process so that root process can merge the result. Slave processes are using standard library’s Map data structure (std::map) since it is very suitable to be used in key,value pair implementations. While

std::map is mapping the data, it also sorts the keys (words in our problem) in lexicographic order whenever a new key is inserted.

- After map and sort procedure finish, slave processes convert std::map to custom struct so that data can be sent via MPI.
- When root process receive the result sent by slave processes, root process merge the results.
- After merge procedure finish, root process prints the result.

5.1.2 Split File Method

In the "SplitFile" method located in FileSplitter.cpp file, the following items are managed.

- This method trims the beginning of the input if there is white space in the beginning of the input.
- Input is divided into n-1 equal partitions. After that, residual value is added to partitions from the beginning of the partition. (Block Distribution)[7]
- The brackets are checked so that it shouldn't divide any word into two. Therefore, it should be on white space.
- If bracket is on white space, it should find the last white space if there are consecutive white spaces.
- If bracket is reached to end of file, all the other brackets will be assigned to end of file. The number of brackets which are equal to end of file will determine the number of passive processes.
- Brackets indicate the boundary of each process.
- Any bracket cannot be less than previous bracket.

5.1.3 Eliminate Passive Processes Method

In the "EliminatePassiveProcesses" method located in ProcessManager.cpp file, brackets will be visited in the reverse order by skipping the last element. If the visited element is equal to last element, it will be removed. This process will be repeated until method finds an element different than last element. Since the bracket list is sorted, method terminate when found an element different than the last element. The number of brackets after this methods will determine the number of active processes.

5.1.4 Map Method

In the "Map" method located in WordCounter.cpp file, every process will convert input string to tokenized format even if it has already been tokenized. In order to tokenize the input, boost library is used. $, .\backslash t \backslash n \backslash$ "- characters are used to separate the input word by word. When program face with these characters, string will be separated into two. After tokenization operation, every word will be transformed to lower letter so that same words shouldn't be counted as two different words because one of them has capital letter, the other hasn't. After that, word will be searched on the Map Data Structure to check whether the word has been found before or not. If it is found, it's value will be incremented by one; else new key will be generated by Map Data Structure and it's value will be set to 1.

5.1.5 Convert Map to Struct Method

In the "ConvertMaptoStruct" method located in WordCounter.cpp file, Map Object will be converted to struct which is defined to use in MPI routines so that every processes communicate with each other.

5.1.6 Merge Method

In the "Merge" method located in WordCounter.cpp file, root process will merge the received struct which is defined to use in MPI routines. Root process traversing on the struct array and check that whether map object has the word. If it has the word, it will increase the count of the word with a value of the count variable of the struct. If it hasn't the word, it will create a new key and assign its value to count variable of the word. Map object will locate the new word in lexicographic order.

5.1.7 Print Result Method

In the ”Merge” method located in WordCounter.cpp file, root process will traverse on the map object and print the result.

5.2 Data Structures

5.2.1 Map Data Structure

Standard Library Map Data Structure is very suitable to be used for this task. There is a key-value pair and the keys should be sorted in lexicographic order. Every process will hold a map object and when any process encountered with a word, first it will use the ”find” method to check that whether the word has been encountered before. If it has been encountered before, process will increase the value of that key. If it hasn't been encountered before, it will create a new key and assign 1 for the value of that key.

Since keys are stored in sorted order, we don't need to apply any sort algorithm.[5]

5.2.2 Custom Struct Data Structure

Custom struct is defined to send the data between processes since Open MPI doesn't support any STL Data Structures except vector. As it is stated in the description of the project, the custom struct is composed of two variables. One of them is a character array, the other is unsigned integer. Character array is 50-byte long since the longest word in English is 45-byte long as it is stated in the Section 2. The unsigned integer variable will be used to store the number of occurrences of the word.

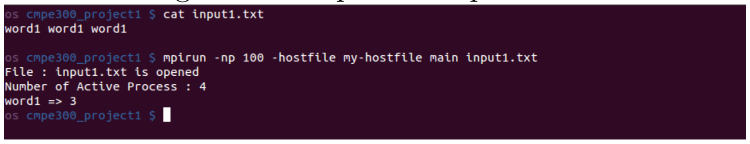
6 Examples

In this section, the execution of the program will be explained on different examples.

6.1 Example 1

Let's assume that we have an input file as shown in Figure 4. The output of the program can also be seen in the Figure 4.

Figure 4: Output for Input File 1



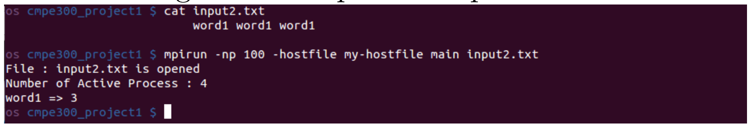
```
os cmpe300_project1 $ cat input1.txt
word1 word1 word1

os cmpe300_project1 $ mpirun -np 100 -hostfile my-hostfile main input1.txt
File : input1.txt is opened
Number of Active Process : 4
word1 => 3
os cmpe300_project1 $
```

As it can be seen in the Figure 4, although 100 processes was created by giving as input argument, number of active processes was displayed as 4. It is because of the fact that input is too small to divide on 100 processes. 4 process including root process will be enough to process this input.

6.2 Example 2

Let's assume that we have an input file which is very similar with input file used in Example 1. The only difference is that there are too many space in the beginning and end of the file. The root process will trim these spaces and therefore the number of active process will be same as Example 1 as shown in Figure 5.

Figure 5: Output for Input File 2

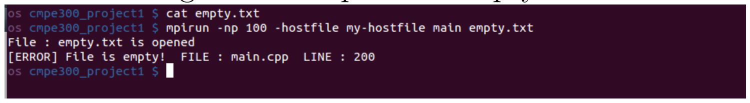


```
os cmpe300_project1 $ cat input2.txt
                          word1 word1 word1

os cmpe300_project1 $ mpirun -np 100 -hostfile my-hostfile main input2.txt
File : input2.txt is opened
Number of Active Process : 4
word1 => 3
os cmpe300_project1 $
```

6.3 Example 3

Let's assume that we have an empty file and it is given to program as input. The program will terminate by giving an error message as shown in Figure 6.

Figure 6: Output for Empty File

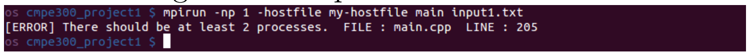


```
os cmpe300_project1 $ cat empty.txt
os cmpe300_project1 $ mpirun -np 100 -hostfile my-hostfile main empty.txt
File : empty.txt is opened
[ERROR] File is empty!  FILE : main.cpp  LINE : 200
os cmpe300_project1 $
```

6.4 Example 4

Let's assume that user chooses one process as the number of process. The program will terminate by giving an error message as shown in Figure 7.

Figure 7: Output for One Process



```
os cmpe300_project1 $ mpirun -np 1 -hostfile my-hostfile main input1.txt
[ERROR] There should be at least 2 processes.  FILE : main.cpp  LINE : 205
os cmpe300_project1 $ 
```

7 Improvements and Extensions

There are a couple of topics we need to consider for further use of this program. These topics will be discussed in this section.

In the communication between slave and root processes, there is a hierarchy in the slave processes. If slave process whose rank is smaller than some processes fails, the system faces with deadlock since root process will wait the slave process which fails and root process cannot take data which is sent by other processes whose rank is higher.

The length of the word is determined statically by referencing Oxford Dictionary. [2] However, in case program is faced with longer word, full word cannot be taken. For further use, it is better to decide word length dynamically. The other disadvantage for static allocation is that program sending redundant information for smaller words. This choice causes inefficient use of the bandwidth and fill the bandwidth with redundant characters.

8 Difficulties Encountered

During the design of the project, I faced with some difficulties since there is not any method to send map object via MPI. In boost library, it is allowed to send vector in C++, however other STL data structures cannot be sent via boost library.

Other difficulty which I encountered is that it is very difficult to count compound words or abbreviations. For example, supposing that we have an input "I'm self-confident.". This input can be counted different ways. I can split the compound words into two or I can take whole compound word. The another point in this sentence is that how I should count "I'm" word. It is up to programmer. Therefore, different programs can take same input and give different results.

9 Conclusion and Assessment

In this report, the design of Map Reduce for counting word occurences in a file was explained in detail by expressing the design choices. The source code can be found in the Appendices.

References

[1] https://www.slideshare.net/imcinstitute/big-data-hadoop-using-amazon- elastic-mapreduce-handson-labs. Accessed at Nov 13, 2017

[2] https://en.oxforddictionaries.com/explore/what-is-the-longest-english-word

[3] http://www.open-mpi.org/

[4] http://lsi.ugr.es/jmantas/pdp/ayuda/datos/instalaciones/Install_OpenMPI_en.pdf

[5] http://www.cplusplus.com/reference/map/map/map/

[6] https://www.open-mpi.org/faq/?category=running#oversubscribing

[7] RS/6000 SP: Practical MPI Programming, IBM

Appendices

Main.cpp

```
#include <iostream>
#include <fstream>
#include <cstring>
#include <new>
#include <map>
#include "mpi.h"
#include "Datatype.h"
#include "FileSplitter.h"
#include "ProcessManager.h"
#include "WordCounter.h"

int main(int argc, char** argv)
{
        int  nNumberOfProcess, nProcessRank;

        // initialize MPI
        MPI_Init(&argc,&argv);

        // get number of tasks
        MPI_Comm_size(MPI_COMM_WORLD,&nNumberOfProcess);

        // get my rank
        MPI_Comm_rank(MPI_COMM_WORLD,&nProcessRank);

        MPI_Datatype stMPIWordMapType, pstMPIWordMapOldTypes
           [2];   // required variables
        int pnMPIWordMapBlockCounts[2];

        // MPI_Aint type used to be consistent with syntax of
            MPI_Type_extent routine
        MPI_Aint pnMPIWordCountOffsets[2],
           nMPIWordCountExtent;

        // setup description of the
           MAX_CHAR_ALLOWABLE_IN_WORD MPI_CHAR field pcWord
        pnMPIWordCountOffsets[0] = 0;
        pstMPIWordMapOldTypes[0] = MPI_CHAR;
        pnMPIWordMapBlockCounts[0] =
           MAX_CHAR_ALLOWABLE_IN_WORD;

        // setup description of the 1 MPI_UNSIGNED field
```

```
   nWordCount
// need to first figure offset by getting size of
   MPI_CHAR
MPI_Type_extent(MPI_CHAR, &nMPIWordCountExtent);
pnMPIWordCountOffsets[1] = MAX_CHAR_ALLOWABLE_IN_WORD
    * nMPIWordCountExtent;
pstMPIWordMapOldTypes[1] = MPI_UNSIGNED;
pnMPIWordMapBlockCounts[1] = 1;

// define structured type and commit it
MPI_Type_struct(2, pnMPIWordMapBlockCounts,
   pnMPIWordCountOffsets, pstMPIWordMapOldTypes, &
   stMPIWordMapType);
MPI_Type_commit(&stMPIWordMapType);

try
{
        if(nProcessRank == MASTER_PROCESS_RANK)
    {
                if(nNumberOfProcess < 2)
                        throw
                           EN_EXCEPTION_PROCESS_NUMBER
                           ;

                PRINT_DEBUG_MESSAGE("Number of
                   Process : " << nNumberOfProcess);

                std::ifstream cInputFile;

                cInputFile.open(argv[1], std::ios::in
                     | std::ios::ate);

                if(!cInputFile.is_open())
                {
                        throw
                           EN_EXCEPTION_FILE_NOT_FOUND
                           ;
                }
                else
                {
                        PRINT_INFO_MESSAGE("File : "
                           << argv[1] << " is opened
                           ");
                        std::streampos nFileBegin,
                           nFileEnd;
```

```
nFileEnd = cInputFile.tellg()
   ;
cInputFile.seekg (0, std::ios
   ::beg);
nFileBegin = cInputFile.tellg
   ();
std::size_t nInputFileSize =
   nFileEnd - nFileBegin;

if(nInputFileSize == 0)
{
        cInputFile.close();
        throw
           EN_EXCEPTION_FILE_EMPTY
           ;
}

char* pcFileBuffer = new (std
   ::nothrow) char[
   nInputFileSize];
if(pcFileBuffer == NULL)
        throw
           EN_EXCEPTION_OUT_OF_MEMORY
           ;

cInputFile.read(pcFileBuffer,
    nInputFileSize);

std::vector<unsigned int>
   cBrackets(nNumberOfProcess
   , 0);
FileSplitter::SplitFile(
   pcFileBuffer,
   nInputFileSize,
   nNumberOfProcess,
   cBrackets);
ProcessManager::
   EliminatePassiveProcesses(
   cBrackets);
PRINT_INFO_MESSAGE("Number of
    Active Process : " <<
   cBrackets.size());

int nActiveProcessFlag = 0;
```

```
for(unsigned int
   nProcessIndex = 1;
   nProcessIndex <
   nNumberOfProcess;
   nProcessIndex++)
{
        if(nProcessIndex <
           cBrackets.size())
                nActiveProcessFlag
                    = 1;
        else
                nActiveProcessFlag
                    = 0;

        MPI_Send(&
           nActiveProcessFlag
           , 1, MPI_INT,
           nProcessIndex, 1,
           MPI_COMM_WORLD);

        unsigned int
           nDataSize =
           cBrackets[
           nProcessIndex] -
           cBrackets[
           nProcessIndex -
           1];

        PRINT_DEBUG_MESSAGE("
           Process 0 is sent
           " << nDataSize <<
           " bytes to process
            " <<
           nProcessIndex);

        MPI_Send(&nDataSize,
           1, MPI_INT,
           nProcessIndex, 1,
           MPI_COMM_WORLD);
        MPI_Send(pcFileBuffer
            + cBrackets[
           nProcessIndex -
           1], nDataSize ,
           MPI_CHAR,
           nProcessIndex, 2,
                    MPI_COMM_WORLD);
        }

        //std::string strPartOfFile(
           pcFileBuffer, cBrackets
           [0]);
        std::map<std::string,
           unsigned int> cWordMap;

        for(unsigned int
           nActiveProcessIndex = 1;
           nActiveProcessIndex <
           cBrackets.size();
           nActiveProcessIndex++)
        {
                int
                   nNumberOfDistinctWord
                   ;
                MPI_Status status;

                MPI_Recv(&
                   nNumberOfDistinctWord
                   , 1, MPI_INT,
                   nActiveProcessIndex
                   ,1, MPI_COMM_WORLD
                   , &status);

            // Allocate a buffer to
               hold the incoming
               numbers
            WordMap* pstWordMap = new
                (std::nothrow)
               WordMap[
               nNumberOfDistinctWord
               ];
                if(pstWordMap == NULL
                   )
                        throw
                           EN_EXCEPTION_OUT_OF_MEMORY
                           ;

            // Now receive the
               message with the
               allocated buffer
```

```
                                MPI_Recv(pstWordMap,
                                   nNumberOfDistinctWord,
                                    stMPIWordMapType,
                                   nActiveProcessIndex,
                                   2, MPI_COMM_WORLD, &
                                   status);
                                PRINT_DEBUG_MESSAGE("
                                   Process " <<
                                   nProcessRank << "
                                   received " <<
                                   nNumberOfDistinctWord
                                   << " words.");

                                 WordCounter::Merge(
                                    cWordMap,
                                    pstWordMap,
                                    nNumberOfDistinctWord
                                    );

                                 delete[] pstWordMap;
                        }

                        WordCounter::PrintResult(
                           cWordMap);

                        delete[] pcFileBuffer;

                        cBrackets.clear();

                        cWordMap.clear();
                }

                cInputFile.close();

        }
        else
        {
                int nActiveProcessFlag;
                int nReceivedDataSize;
                MPI_Status status;

                MPI_Recv(&nActiveProcessFlag, 1,
                   MPI_INT, 0,1, MPI_COMM_WORLD, &
                   status);
```

```
if(nActiveProcessFlag)
{
        MPI_Recv(&nReceivedDataSize,
            1, MPI_INT, 0,1,
            MPI_COMM_WORLD, &status);

    // Allocate a buffer to hold the
       incoming numbers
    char* pcBuffer = new (std::
       nothrow) char[
       nReceivedDataSize+1];
        if(pcBuffer == NULL)
                throw
                    EN_EXCEPTION_OUT_OF_MEMORY
                    ;
        memset(pcBuffer,0,
            nReceivedDataSize+1);
    // Now receive the message with
       the allocated buffer
    MPI_Recv(pcBuffer,
       nReceivedDataSize, MPI_CHAR,
       0, 2, MPI_COMM_WORLD, &status)
       ;
    PRINT_DEBUG_MESSAGE("Process " <<
        nProcessRank << " received "
       << nReceivedDataSize << "
       bytes.");

        pcBuffer[nReceivedDataSize] =
             '\0';
    std::string strPartOfFile(
       pcBuffer);
        std::map<std::string,
            unsigned int> cWordMap =
            WordCounter::Map(
            strPartOfFile);

        WordMap* pstWordMap = new (
            std::nothrow) WordMap[
            cWordMap.size()];
        if(pstWordMap == NULL)
                throw
                    EN_EXCEPTION_OUT_OF_MEMORY
                    ;
```

```
                                memset(pstWordMap, 0, sizeof(
                                   pstWordMap));
                                WordCounter::
                                   ConvertMaptoStruct(
                                   cWordMap, pstWordMap);

                                int nNumberOfDistinctWord =
                                   cWordMap.size();
                                PRINT_DEBUG_MESSAGE("Process
                                   " << nProcessRank <<" is
                                   sent " <<
                                   nNumberOfDistinctWord << "
                                    word to process 0");
                                MPI_Send(&
                                   nNumberOfDistinctWord, 1,
                                   MPI_INT,
                                   MASTER_PROCESS_RANK, 1,
                                   MPI_COMM_WORLD);
                                MPI_Send(pstWordMap,
                                   nNumberOfDistinctWord ,
                                   stMPIWordMapType,
                                   MASTER_PROCESS_RANK, 2,
                                   MPI_COMM_WORLD);


                                delete[] pstWordMap;
                        delete[] pcBuffer;
                }
        }
}
catch(ExceptionType eExceptionType)
{
        switch(eExceptionType)
        {
                case EN_EXCEPTION_OUT_OF_MEMORY:
                {
                        PRINT_ERROR_MESSAGE("Unable
                           to allocate buffer");
                }
                break;
                case EN_EXCEPTION_FILE_EMPTY:
                {
                        PRINT_ERROR_MESSAGE("File is
                           empty!");
                }
```

```
                        break;
                        case EN_EXCEPTION_PROCESS_NUMBER:
                        {
                                PRINT_ERROR_MESSAGE("There
                                   should be at least 2
                                   processes.");
                        }
                        break;
                        case EN_EXCEPTION_FILE_NOT_FOUND:
                        {
                                PRINT_ERROR_MESSAGE("File : "
                                    << argv[1] << " cannot be
                                    opened");
                        }
                        break;
                        default:
                        break;
                }

                for(unsigned int nProcessIndex = 1;
                   nProcessIndex < nNumberOfProcess;
                   nProcessIndex++)
                {
                        int nActiveProcessFlag = 0;
                        MPI_Send(&nActiveProcessFlag, 1,
                           MPI_INT, nProcessIndex, 1,
                           MPI_COMM_WORLD);
                }

        }

        // free datatype when done using it
        MPI_Type_free(&stMPIWordMapType);

        // done with MPI
        MPI_Finalize();



}
```

Datatype.h

```
#ifndef __DATATYPE_H__
#define __DATATYPE_H__

#include <iostream>

#define PRINT_DEBUG_MESSAGE(MESSAGE) //std::cout << "[DEBUG]
   " << MESSAGE << "  FILE : " << __FILE__ << "  LINE : " <<
   __LINE__ << std::endl
#define PRINT_INFO_MESSAGE(MESSAGE) std::cout << MESSAGE <<
   std::endl
#define PRINT_ERROR_MESSAGE(MESSAGE) std::cerr << "[ERROR] "
   << MESSAGE << "  FILE : " << __FILE__ << "  LINE : " <<
   __LINE__ << std::endl

#define MASTER_PROCESS_RANK 0

enum ExceptionType
{
        EN_EXCEPTION_OUT_OF_MEMORY,
        EN_EXCEPTION_FILE_EMPTY,
        EN_EXCEPTION_PROCESS_NUMBER,
        EN_EXCEPTION_FILE_NOT_FOUND
};

#endif
```

FileSplitter.cpp

```
#include "FileSplitter.h"
#include "Datatype.h"


void FileSplitter::SplitFile(char* pcInputFile, unsigned int
   nFileSize, unsigned int nNumberOfProcess, std::vector<
   unsigned int>& cBrackets)
{
        PRINT_DEBUG_MESSAGE("nFileSize : " << nFileSize);
        PRINT_DEBUG_MESSAGE("nNumberOfProcess : " <<
           nNumberOfProcess);

        while(pcInputFile[cBrackets[MASTER_PROCESS_RANK]] ==
           '\t' || pcInputFile[cBrackets[MASTER_PROCESS_RANK
           ]] == ' ' || pcInputFile[cBrackets[
           MASTER_PROCESS_RANK]] == '\n')
        {
                cBrackets[MASTER_PROCESS_RANK]++;
                if(cBrackets[MASTER_PROCESS_RANK] ==
                   nFileSize)
                {
                        PRINT_DEBUG_MESSAGE("File limit is
                           reached.");
                        break;
                }
        }

        if(cBrackets[MASTER_PROCESS_RANK] > 0)
                cBrackets[MASTER_PROCESS_RANK]--;

        unsigned int nSizePerProcess = (nFileSize - cBrackets
           [MASTER_PROCESS_RANK]) / (nNumberOfProcess-1);
        unsigned int nResidualData = (nFileSize - cBrackets[
           MASTER_PROCESS_RANK]) % (nNumberOfProcess-1);

        for(unsigned int nProcessIndex = 1; nProcessIndex <
           nNumberOfProcess; nProcessIndex++)
        {
                cBrackets[nProcessIndex] = nSizePerProcess *
                   nProcessIndex;
                PRINT_DEBUG_MESSAGE("1. Before arranged
                   cBrackets[" << nProcessIndex << "] = " <<
                   cBrackets[nProcessIndex]);
```

```
if(nResidualData > 0)
{
        cBrackets[nProcessIndex]++;
        nResidualData--;
}
PRINT_DEBUG_MESSAGE("2. Before arranged
   cBrackets[" << nProcessIndex << "] = " <<
   cBrackets[nProcessIndex]);

if(cBrackets[nProcessIndex-1] < (nFileSize -
   1))
{
        if(cBrackets[nProcessIndex-1] >=
           cBrackets[nProcessIndex])
                cBrackets[nProcessIndex] =
                   cBrackets[nProcessIndex -
                   1] + 1;
}
else
{
        cBrackets[nProcessIndex] = nFileSize
           - 1;
        continue;
}

PRINT_DEBUG_MESSAGE("1. After arranged
   cBrackets[" << nProcessIndex << "] = " <<
   cBrackets[nProcessIndex]);

while(cBrackets[nProcessIndex] < (nFileSize
   -1))
{
        if(pcInputFile[cBrackets[
           nProcessIndex]] != '\t' &&
           pcInputFile[cBrackets[
           nProcessIndex]] != ' '  &&
           pcInputFile[cBrackets[
           nProcessIndex]] != '\n')
                cBrackets[nProcessIndex]++;
        else
                break;
}

PRINT_DEBUG_MESSAGE("2. After arranged
   cBrackets[" << nProcessIndex << "] = " <<
```

```
                    cBrackets[nProcessIndex]);

                while(cBrackets[nProcessIndex] < (nFileSize
                    -1))
                {
                        if(pcInputFile[cBrackets[
                            nProcessIndex]] == '\t' ||
                            pcInputFile[cBrackets[
                            nProcessIndex]] == ' '  ||
                            pcInputFile[cBrackets[
                            nProcessIndex]] == '\n')
                                cBrackets[nProcessIndex]++;
                        else
                        {
                                cBrackets[nProcessIndex]--;
                                break;
                        }
                }

                PRINT_DEBUG_MESSAGE("3. After arranged
                    cBrackets[" << nProcessIndex << "] = " <<
                    cBrackets[nProcessIndex]);


        }

}
```

FileSplitter.h

```
#ifndef __FILESPLITTER_H__
#define __FILESPLITTER_H__

#include <iostream>
#include <vector>

class FileSplitter
{
        // Attributes
        private:

        public:

        // Methods
        private:

        public:
                static void SplitFile(char* pcInputFile,
                   unsigned int nFileSize, unsigned int
                   nNumberOfProcess, std::vector<unsigned int
                   >& cBracketsList);
};

#endif
```

ProcessManager.cpp

```
#include "ProcessManager.h"

void ProcessManager::EliminatePassiveProcesses(std::vector<
   unsigned int>& cBracketsList)
{
        for(std::vector<unsigned int>::reverse_iterator
            cIterator = cBracketsList.rbegin()+1; cIterator !=
             cBracketsList.rend(); cIterator++)
        {
                if(*cIterator == *(cBracketsList.rbegin()))
                        cBracketsList.erase(--cIterator.base
                           ());
                else
                        break;

        }
}
```

ProcessManager.h

```
#ifndef __PROCESSMANAGER_H__
#define __PROCESSMANAGER_H__

#include <iostream>
#include <vector>

class ProcessManager
{
        // Attributes
        private:

        public:

        // Methods
        private:

        public:
                static void EliminatePassiveProcesses(std::
                   vector<unsigned int>& cBracketsList);
};

#endif
```

WordCounter.cpp

```
#include "WordCounter.h"
#include <algorithm>
#include <boost/tokenizer.hpp>
#include "Datatype.h"
#include <sstream>


std::map<std::string, unsigned int> WordCounter::Map(std::
   string strInput)
{
        boost::char_separator<char> sep(", .\t\n\"-");
        boost::tokenizer< boost::char_separator<char> >
           tokens(strInput, sep);

        std::map<std::string, unsigned int> cWordMap;

        for(boost::tokenizer< boost::char_separator<char> >::
           iterator cIterator=tokens.begin(); cIterator !=
           tokens.end(); ++cIterator)
    {
        std::string strWord(*cIterator);
        std::transform(strWord.begin(), strWord.end(),
           strWord.begin(), ::tolower);
        PRINT_DEBUG_MESSAGE("Word : " << strWord);
                if(cWordMap.find(strWord) == cWordMap.end())
                        cWordMap[strWord] = 1;
                else
                        cWordMap[strWord]++;
    }

    return cWordMap;

}

void WordCounter::ConvertMaptoStruct(const std::map<std::
   string, unsigned int>& cWordMap, WordMap* pstWordMap)
{
        unsigned int nWordMapIndex = 0;
        for(std::map<std::string, unsigned int>::
           const_iterator cIterator=cWordMap.begin();
           cIterator!=cWordMap.end(); ++cIterator)
        {
                std::copy(cIterator->first.begin(), cIterator
                   ->first.end(), pstWordMap[nWordMapIndex].
```

```
                    pcWord);
                pstWordMap[nWordMapIndex].pcWord[cIterator->
                    first.size()] = '\0';
                pstWordMap[nWordMapIndex].nWordCount =
                    cIterator->second;
                nWordMapIndex++;

        }
}

void WordCounter::Merge(std::map<std::string, unsigned int>&
    cWordMap, WordMap* pstWordMap, unsigned int
    nNumberOfDistinctWord)
{
        for(int nWordMapIndex = 0; nWordMapIndex<
            nNumberOfDistinctWord; nWordMapIndex++)
        {
                std::string strWord(pstWordMap[nWordMapIndex
                    ].pcWord);
                if(cWordMap.find(strWord) == cWordMap.end())
                        cWordMap[strWord] = pstWordMap[
                            nWordMapIndex].nWordCount;
                else
                        cWordMap[strWord] += pstWordMap[
                            nWordMapIndex].nWordCount;
        }
}

void WordCounter::PrintResult(const std::map<std::string,
    unsigned int>& cWordMap)
{
        for(std::map<std::string, unsigned int>::
            const_iterator cIterator = cWordMap.begin();
            cIterator!=cWordMap.end(); ++cIterator)
        std::cout << cIterator->first << " => " << cIterator
            ->second << "\n";
}
```

WordCounter.h

```
#ifndef __WORDCOUNTER_H__
#define __WORDCOUNTER_H__

#include <iostream>
#include <cstring>
#include <map>

#define MAX_CHAR_ALLOWABLE_IN_WORD 50

struct WordMap
{
        char pcWord[MAX_CHAR_ALLOWABLE_IN_WORD];
        unsigned int nWordCount;
};

class WordCounter
{
        // Attributes
        private:

        public:

        // Methods
        private:

        public:
                static std::map<std::string, unsigned int>
                   Map(std::string strInput);

                static void ConvertMaptoStruct(const std::map
                   <std::string, unsigned int>& cWordMap,
                   WordMap* pstWordMap);

                static void Merge(std::map<std::string,
                   unsigned int>& cWordMap, WordMap*
                   pstWordMap, unsigned int
                   nNumberOfDistinctWord);

                static void PrintResult(const std::map<std::
                   string, unsigned int>& cWordMap);
};

#endif
```