BOĞAZİÇİ UNIVERSITY

CMPE 300

Analysis of Algorithms

MapReduce Algorithm MPI Programming Project

Mustafa Enes ÇAKIR

2013400105

December 17, 2017

Submitted to Mehmet Köse

Contents

1 Introduction

We are asked to implement MapReduce algorithm using Message Passing Interface (MPI) for calculating word frequencies. Master processor coordinates other slave processors. After this point I will use *master* for master processor and *slaves* for slave processors. *Master* shares data between *slaves* in balanced parts and combines results.
In the given MapReduce algorithm, there 5 basic steps. First of all, *master* reads input file and send raw string lists to *slaves*. Secondly, *slaves* map words with their counts. In this step, we don't look other words, just consider current one. Evidently each word is mapped with count 1. *Slaves* sends mapped word lists to *master* and it combines these list. Thirdly *master* divides mapped word list to *slaves*. They sort corresponding lists lexically and send them back to *master*. Fourthly, *Master* merges sorted list. Finally, it reduces word with summing up their counts.

2 Program Interface

It's a basic command line application written with C++ using Boost MPI library. For compiling and running application, Boost MPI and Open MPI have to be installed. Boost libraries have to be added to *mpic++* that is Open MPI C++ compiler. *Figure 1* shows compile command in detail.

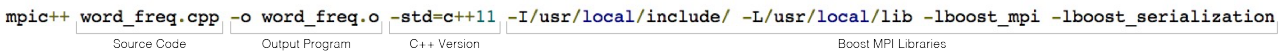

Figure 1: Compilation Command

Compiled code can be executed with *mpirun* command with given number of processor. Application takes 2 parameters. First one is input file, and second one is output file. If output file isn't provides, result printed to console.

Figure 2: Run Command

3 Program Execution

The command line application takes 2 arguments. First one is input file and it's required. Second one is output file and it's optinal. If no file is provided, application prints out output to console. It calculates frequencies of words in given input file, and print words with their frequency in dictionary order. You can see sample execution of application in *Figure 3*.

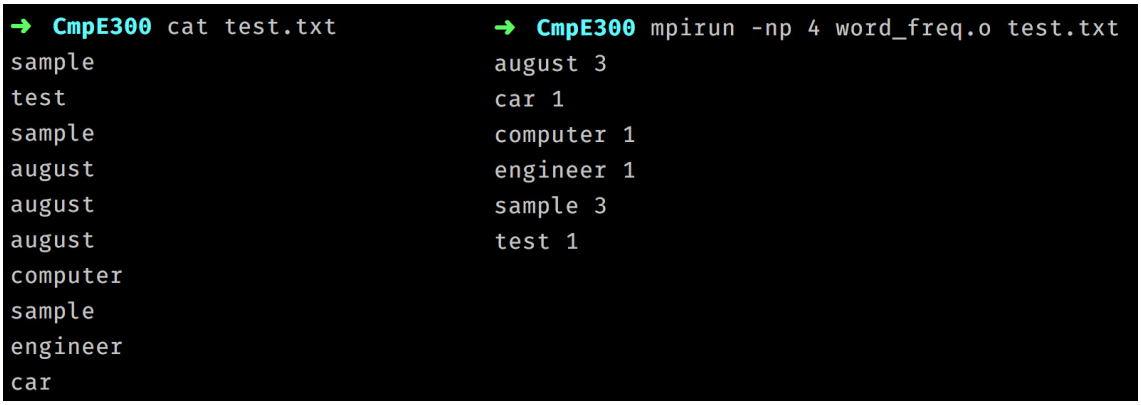

Figure 3: Program Execution with sample input

4 Input and Output

The input has one word on each line. In the other word, it's in tokenized format and has no limitations.

```
sample
test
sample
august
august
august
computer
sample
engineer
car
```

Listing 1: Sample input

Output has one word and it's count on each line in lexical order. It might be in file or on console. If output file name is provided, it prints out to file.

```
august 3
car 1
computer 1
engineer 1
sample 3
test 1
```

Listing 2: Sample output of the sample input

5 Program Structure

The application is developed with C++ with Boost MPI. I used *Word* struct for data representation. I choose rank 0 processor as *master*.

5.1 Reading and Dividing

Master checks arguments of application. If no input file is given, it raises an error. If an output file is given, changes standard output with file It reads input file line by line and pushes word strings to a vector. *Master* divides this vector using index modulo. For example words indexed by 2, 5, 8, 11 goes to *slave* rank with 2 in world that is sized 4.

5.2 Mapping

Slaves get their string subvectors and constructs *Word* structs with count 1 for every word string. Lastly *slaves* send mapped words to *master*. *Master* combines coming *Word* vectors.

5.3 Sorting

Master shares this mapped vector to *slaves* again. *Slaves* sort received subvectors using *std::sort* and send back to *master*.

5.4 Merging

So received vectors by master are sorted now. As the last step of merge sort, *master* merges that sorted vectors with *std::merge*. For this merge opera-

tion I used *temp_words* vector. After merging to *temp_words*, I swapped *temp_words* with *sorted_words*.

5.5 Reducing

Master traverses *sorted_words* vector and reduces it. If the current *Word* is same as the last word, increase count of the last word. Else, it changes the last word with current one.

5.6 Word Struct

The main data structure in application is *Word* struct. It's combination of word's text and it's count. Unusual part of this code is *serialize* method. It's transform the word to format that can be send via MPI.

```
1  struct Word {
2  public:
3      string text;
4      int count;
5
6  private:
7      friend class boost::serialization::access;
8
9      /**
10      * Serializes Word struct for sending via Boost MPI
11      */
12     template<class Archive>
13     void serialize(Archive &ar, const unsigned int version) {
14         ar & text;
15         ar & count;
16     }
17
18 };
```

Code 1: Code of Word struct

6 Examples

The application executed with sample input *Listing 1*. It gives sample output *Listing 2*.

```
➜ CmpE300 cat test.txt
sample
test
sample
august
august
august
computer
sample
engineer
car
```

```
➜ CmpE300 mpirun -np 4 word_freq.o test.txt
august 3
car 1
computer 1
engineer 1
sample 3
test 1
```

Figure 4: Program Execution with sample input

7 Improvements and Extensions

The algorithm that is given in project description has some unnecessary steps. At the first step all word mapped with 1. I think while we are sending to *slaves* from *master*, we can mapped with 1. Also before sending them back to *master*, we can sort them. Because when divide mapped ones to *slaves*, they get some words with words that come as raw string.

8 Difficulties Encountered

At the beginning, I tried to develop application with *Open MPI* library. But it's very low level library. Programming interface isn't very modern. I don't like to get output via parameter not returning variable. Also it has limited primitives types that you can send via message. Boost MPI has more modern interface. It's more useful. Learning 4-5 functions was enough for this project. Also *std::merge* has very interesting signature. I had difficulties while using it.

9 Conclusion

We are learning *Parallel Programming* paradigm in *CmpE344*, *CmpE322* and *CmpE300*. But all of them are theoretical. This project was a great chance to see real applications of it. Now, more stones fit to holes.

10 Appendices

Next lines contain source code of application.

```
#include <boost/mpi.hpp>
#include <iostream>
#include <vector>
#include <fstream>

/**
 * It's rank of the master processor
 */
#define MASTER_PROCESSOR 0

using namespace std;
namespace mpi = boost::mpi;

/**
 * Word struct keeps information of string and it's count
 */
class Word {
public:
    string text;
    int count;

    /**
     * Default constructor
     */
    Word() {
        text = "";
        count = 0;
    }

    /**
     * Default constructor
     */
    Word(string text_, int count_) {
        text = text_;
        count = count_;
    }

```

```
38      /**
39       * Override equal operator
40       * @param w Word to compare with this one
41       * @return true if text of the this word is equal to
   ↪  other one
42       */
43      bool operator==(Word w) const {
44          return text == w.text;
45      }
46
47  private:
48      friend class boost::serialization::access;
49      /**
50       * Serializes Word struct for sending via Boost MPI
51       */
52      template<class Archive>
53      void serialize(Archive &ar, const unsigned int version) {
54          ar & text;
55          ar & count;
56      }
57  };
58
59  /**
60   * Compare two words depends on their text
61   */
62  struct wordComparator {
63      bool operator()(Word w1, Word w2) {
64          return w1.text < w2.text;
65      }
66  };
67
68  /**
69   * Entry point of my application
70   */
71  int main(int argc, char *argv[]) {
72      mpi::environment env(argc, argv);
73      mpi::communicator world;
74
75      /**
76       * Master processor work
77       */
```

```
78      if (world.rank() == MASTER_PROCESSOR) {
79          // Contains raw strings
80          vector<string> strings;
81
82          // Checks arguments count for input and output file
83          if (argc < 2) {
84              // If no input file name is give, raise an error
85              cout << "Run the code with the following command:
                ↪  mpirun --oversubscribe -np [processor_count]
                ↪  main.o [input_file] [output_file]?" << endl;
86              return 1;
87          } else {
88              // Read raw strings
89              ifstream in(argv[1]);
90              string line;
91              while (in >> line) {
92                  strings.push_back(line);
93              }
94              in.close();
95              // If output file is given, print out to it
96              if (argc == 3) {
97                  freopen(argv[2], "w", stdout);
98              }
99          }
100
101         // Divide raw string array and send to slaves
102         for (int i = 1; i < world.size(); i++) {
103             vector<string> substrings;
104             for (int j = 0; j < strings.size(); j++) {
105                 if (j % (world.size() - 1) == i - 1)
                    ↪  substrings.push_back(strings[j]);
106             }
107             world.send(i, 1, substrings);
108         }
109         // Collect mapped word struct vectors from slaves
110         vector<Word> words;
111         for (int i = 1; i < world.size(); ++i) {
112             vector<Word> subwords;
113             world.recv(i, 2, subwords);
114             words.insert(words.end(), subwords.begin(),
                ↪  subwords.end());
```

```
115         }
116         // Divide mapped word vector and send to slaves
117         for (int i = 1; i < world.size(); i++) {
118             vector<Word> subwords;
119             for (int j = 0; j < words.size(); j++) {
120                 if (j % (world.size() - 1) == i - 1)
                    ↪  subwords.push_back(words[j]);
121             }
122             world.send(i, 3, subwords);
123         }
124         // Collect sorted mapped word struct vectors from
            ↪  slaves and merge
125         vector<Word> sorted_words;
126         for (int i = 1; i < world.size(); ++i) {
127             vector<Word> subwords;
128             world.recv(i, 4, subwords);
129             vector<Word> temp_words(sorted_words.size() +
                ↪  subwords.size());
130             // Merge sorted vectors that come from slaved
131             merge(sorted_words.begin(),  sorted_words.end(),
                ↪  subwords.begin(), subwords.end(),
                ↪  temp_words.begin(), wordComparator());
132             swap(sorted_words, temp_words);
133         }
134         // Reduce words vector and sum up their counts
135         vector<Word> reduced_words;
136         Word last_word = sorted_words.front();
137         for (int i = 1; i < sorted_words.size() + 1; i++) {
138             if (last_word == sorted_words[i]) {
139                 // If the last one is same word, increase
                    ↪  count
140                 last_word.count++;
141             } else {
142                 // If it's different word, push it to reduced
                    ↪  one and change last word
143                 reduced_words.push_back(last_word);
144                 last_word = sorted_words[i];
145             }
146         }
147         // Print out strings and their counts
148         for (int i = 0; i < reduced_words.size(); i++) {
```

```
149             cout << reduced_words[i].text << " " <<
                ↪  reduced_words[i].count << endl;
150         }
151     } else {
152         /**
153          * Slave processors works
154          */
155         // Receive raw string vector
156         vector<string> strings;
157         world.recv(MASTER_PROCESSOR, 1, strings);
158         // Map string to Word with count
159         vector<Word> words;
160         for (int i = 0; i < strings.size(); i++) {
161             words.push_back(Word(strings[i], 1));
162         }
163         // Sent back to mapped struct to master
164         world.send(MASTER_PROCESSOR, 2, words);
165         // Receive mapped vectors to sort
166         vector<Word> words_to_sort;
167         world.recv(MASTER_PROCESSOR, 3, words_to_sort);
168         // Sort subvectors
169         sort(words_to_sort.begin(), words_to_sort.end(),
            ↪  wordComparator());
170         // Send sorted vectors to master
171         world.send(MASTER_PROCESSOR, 4, words_to_sort);
172     }
173     return 0;
174 }
```

Code 2: Source Code of Application