Application Project - MapReduce

CMPE300(Analysis of Algorithms) Fall 2017

Dilara Gökay - 2014400102

December 17, 2017

Submitted person: Mehmet Köse

Department of Computer Engineering, Boğaziçi University

Introduction

Problem of this project is to implement MapReduce algorithm to count the word occurences in a file.

MapReduce is a programming model for processing and generating large data sets. User specifies a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model. [1]

I used Boost.MPI library for implementing MapReduce. Boost is a collection of free, peer-reviewed C++ libraries.[2]

Program Interface

In order to be able to compile and run the program, user has to install Boost. We assume that user already installed gcc.

Installation of Boost in OSX(also the OS that I use):

1. (If Homebrew is already installed, skip to 2) Install Homebrew by following the steps here: https://brew.sh/
2. Open Terminal and install Boost by the following command:

   ```
   brew install boost boost-mpi
   ```

3. Go to the directory where program exists from Terminal. Compile program by the following command:

   ```
   mpic++ -std=c++11 main.cpp -o main -L/usr/local/lib
   -I/usr/local/include/ -lboost_mpi -lboost_serialization
   ```

   where main.cpp is the name of the program and main is the name of exeuctable.

4. Run program by the following command:

   ```
   mpirun --oversubscribe -np 10 main
   ```

   where 10 can be replaced by any number of processors wished.

Program Execution

User can use this program to find out the repetition of each word in a speech.

User should have a tokenized(ie. each word seperated by space or new line) speech file as input. Hence, punctuation marks in the original speech text

should not be included in tokenized file. The program counts all words in the given text file and writes all words and their number of repetitions in output file.

User should follow,

- Program Interface section to compile and run program.
- Input and Output section to be able to supply correct format of input to program.

Input and Output

Input

- The name of input file should be speech_tokenized.txt
- The input file should be in the same directory with program file.
- Input file should contain all words tokenized (ie. seperated by space or new line).

Output

- The name of the generated output file is output.txt
- User can find output file in the same directory with program file.
- Each line of output file contains one word and one integer respectively.

Program Structure

Data Structures

I used "token" class. It contains a word which as appeared in text file and its number of occurences in the text.

Execution of Program

Here, I explain execution of program step by step and introduce the algorithms and data structures I used meanwhile.

1. Creation of MPI environment and communicator.

```
mpi::environment env(argc, argv);
mpi::communicator world;
```

2. Read speech_tokenized.txt and creates token for each of word with zero repetition. Handled by ReadSpeechFile function.
3. Calculating input size: Done by CalculateNumTokens function. Calculates number of tokens that should be given to each processor. Aim is to distribute number of tokens in each processor evenly so that program runs more efficiently.
4. Split the input and send them to slaves: Each slave gets a subvector according to the calculated input size in step 2. If input size cannot be distributed evenly then last processor gets less input.
5. Slaves map words and send back to master: Slaves make the count of each token 1 and send back to master.

```
// Step 2: Slaves map the words and send it back to master
world.recv(0, 0, tokens);
for (int i = 0; i < tokens.size(); i++) {
    tokens[i].count = 1;
}
world.send(0, 0, tokens);
```

6. Split the input again and send them to slaves to be sorted
7. Each slave sorts the tokens in its subvector according to alphabetical order. Slaves send back the sorted subvectors to master. I used std::sort in order to sort subvectors.
8. Master appends all subvectors received from slaves.
9. Master sorts the appended vector. I used std::sort in order to sort subvectors.
10. The master process reduces the list. All words were stored in ”token” with one repetition. I used std::map to map words(string) to their number of repetitions(int)
11. Write reduced map to output.txt: Handled by WriteOutputFile function.

Examples

- Input:
  dilara
  cmpe300
  dilara

algorithms
a
b
analysis
of
algorithms

- Output:
  a 1
  algorithms 2
  analysis 1
  b 1
  cmpe300 1
  dilara 2
  of 1

Improvements and Extensions

- I was planning to user OpenMPI in order to implement MapReduce. But then I realized that using Boost library is a more comfortable solution.
- I could have write program in a more readable way. For example I know that there are some parts which I can place into a function.

Difficulties Encountered

- Installing Boost was a pretty tough precedure. At first, I installed OpenMPI but then I realized that Boost would ease things for me a lot. But while trying to install it some conflits occured. I couldn't be able to solve these conflicts for a while. This stressed me a lot. Quite a lot. In the end, I realized that I was solving the conflict by using the wrong command.
- We all learned what MPI is especially in Operating Systems(CMPE322) course. But it was my first time implementing this interface in a program. Also, it took my a while to understand how Boost.MPI works. For example, I considered and searched if the communication between slaves and master is synchronized. It is synchronized in Boost.MPI. Otherwise, I should have added barrier between communication.

Conclusion

I finished all tasks that are required successfully. The program works as it should.

Appendices

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

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

class token
{
private:
    friend class boost::serialization::access;

    template<class Archive>
    void serialize(Archive &ar, const unsigned int version)
    {
        // The word which appeared in the text
        ar & word;
        // The number of occurences of this word
        ar & count;
    }

public:
    std::string word;
    int count;

    token() {};
    token(string w, int c) :
            word(w), count(c)
    {}
};

struct less_than_key
{
    inline bool operator() (const token& token1, const token& token2
    )
    {
        return token1.word < token2.word;
    }
};

bool IsFileEmpty(ifstream& file) {
```

```
45      return file.peek() == ifstream::traits_type::eof();
46  }
47
48  // Reads speech_tokenized.txt and creates token for each of word
        with zero repetition
49  vector<token> ReadSpeechFile(){
50      vector<token> tokens;
51      ifstream in_file("./speech_tokenized.txt");
52      if(IsFileEmpty(in_file)) {
53          ofstream output;
54          output.open ("output.txt");
55          output << "speech_tokenized.txt is empty." << endl;
56          output.close();
57          exit(3);
58      }
59      if(!in_file) {
60          ofstream output;
61          output.open ("output.txt");
62          output << "Cannot open speech_tokenized.txt." << endl;
63          output.close();
64          exit(2);
65      }
66      if(in_file.is_open()) {
67          string word;
68          while (in_file >> word) {
69              tokens.push_back(token(word, 0));
70          }
71      }
72      in_file.close();
73      return tokens;
74  }
75
76  // Calculates the number of tokens that will be sent to each slave
77  int CalculateNumTokens (vector<token> tokens, int num_processors){
78      int input_size = tokens.size() / (num_processors - 1);
79      // If all tokens are not distributed to slaves evenly, then the
        last slave will receive less token
80      // This rule is implemented in main function
81      if (tokens.size() % (num_processors - 1) != 0) {
82          input_size++;
83      }
84      return input_size;
85  }
86
87  // Returns reduced map
88  map<string, int> Reduce(vector<token> subsorted_tokens) {
89      std::map<std::string, int> reduced_map;
90      for(int i = 0; i < subsorted_tokens.size(); i++) {
91          if(reduced_map.count(subsorted_tokens[i].word) > 0) {
92              reduced_map[subsorted_tokens[i].word] = reduced_map[
        subsorted_tokens[i].word] + 1;
93          }
94          else {
95              reduced_map[subsorted_tokens[i].word] = 1;
```

```
96          }
97      }
98      return reduced_map;
99  }
100
101 // Writes reduced map to output.txt
102 void WriteOutputFile(std::map<std::string, int> reduced_map) {
103     ofstream output;
104     output.open ("output.txt");
105     map<std::string, int>::iterator it;
106     for (it = reduced_map.begin(); reduced_map.end() != it; it++) {
107         output << (*it).first << " " << (*it).second << endl;
108     }
109     output.close();
110 }
111
112 int main(int argc, char *argv[])   {
113     mpi::environment env(argc, argv);
114     mpi::communicator world;
115     int num_processors = world.size();
116
117     if(world.rank() == 0) {
118         /* MASTER */
119         // Step 1: Split the input and send them to slaves
120         std::vector<token> tokens = ReadSpeechFile();
121         int input_size = CalculateNumTokens(tokens, num_processors);
122
123         if(input_size < num_processors - 1) {
124             num_processors = input_size + 1;
125             input_size = CalculateNumTokens(tokens, num_processors);
126         }
127
128         for (int i = 1; i < num_processors; i++){
129             int start_index = (i - 1) * input_size;
130             int end_index = i * input_size;
131             // If input size is larger than the last slave can
    receive
132             if(tokens.size() < end_index) {
133                 end_index = tokens.size();
134             }
135             vector<token> newTok(tokens.begin() + start_index,
    tokens.begin() + end_index);
136             world.send(i, 0, newTok);
137         }
138
139         // Step 3: Split the input again and send them to slaves to
    be sorted
140         for (int i = 1; i < num_processors; i++) {
141             vector<token> newTok;
142             world.recv(i, 0, newTok);
143             world.send(i, 0, newTok);
144         }
145         // Receive sorted vectors and append them to
    subsorted_tokens
```

```
146         vector<token> subsorted_tokens;
147         for (int i = 1; i < num_processors; i++) {
148             vector<token> newTok;
149             world.recv(i, 0, newTok);
150             for (int j = 0; j < newTok.size(); j++) {
151                 subsorted_tokens.push_back(newTok[j]);
152             }
153         }
154         // Sort subsorted_tokens. Now it is fully sorted
155         sort(subsorted_tokens.begin(), subsorted_tokens.end(),
    less_than_key());
156 
157         // Step 4: The master process reduces the list
158         map<string, int> reduced_map = Reduce(subsorted_tokens);
159 
160         WriteOutputFile(reduced_map);
161     }
162     else {
163         /* SLAVE */
164         std::vector<token> tokens;
165         // Step 2: Slaves map the words and send it back to master
166         world.recv(0, 0, tokens);
167         for (int i = 0; i < tokens.size(); i++) {
168             tokens[i].count = 1;
169         }
170         world.send(0, 0, tokens);
171         world.recv(0, 0, tokens);
172         // Sort tokens
173         std::sort(tokens.begin(), tokens.end(), less_than_key());
174         world.send(0, 0, tokens);
175     }
176     return 0;
177 }
```

References

1. Dean, Ghemawat. *"MapReduce: Simplified Data Processing on Large Clusters"*
2. *https://svn.boost.org/trac10/*. Accessed at Dec 17, 2017