浙江大學

Advanced Data Structures and Algorithm Analysis

Laboratory Projects 7

Map Reduce

Group 15

聂俊哲 张琦 张溢弛

Date: 2020-05-30

Content

Chapter 1: Introduction to Map Reduce

1.1 Problem Description

1.1.1 Introduce

- In this project, we will introduce the framework of Map-Reduce and implement a MapReduce program to count the appearance of each word in a set of documents, which is a classic project in big-data called '"word-count"

1.1.2 Input format

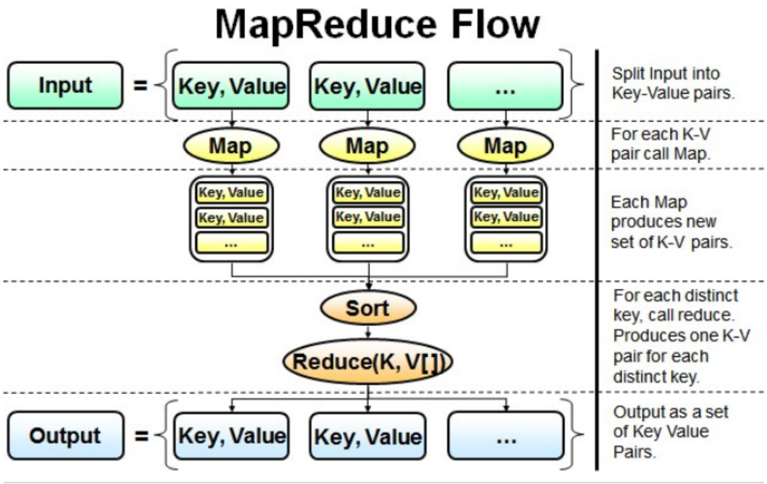
- A set of txt files in the folder "data", no need to input the data from keyboard and stdin.

1.1.3 Output format

- A map of words with their appearance time.
- The total number of the words.
- The running time of the program

1.2 Introduce to the framework of Map-Reduce

- MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.
- A MapReduce Program includes a **map procedure** and a **reduce procedure**
    - map procedure performs filtering and sorting, which means making a pre-process of the big size data.
    - reduce procedure performs a summay operation which means counting the number of each word in this project.
- Another way to make a summary of MapReduce in 5 steps
    - **Prepare the Map() input** – the "MapReduce system" designates Map processors, assigns the input key *K1* that each processor would work on, and provides that processor with all the input data associated with that key.
    - **Run the user-provided Map() code** – Map() is run exactly once for each *K1* key, generating output organized by key *K2*.
    - **"Shuffle" the Map output to the Reduce processors** – the MapReduce system designates Reduce processors, assigns the *K2* key each processor should work on, and provides that processor with all the Map-generated data associated with that key.
    - **Run the user-provided Reduce() code** – Reduce() is run exactly once for each *K2* key produced by the Map step.
    - **Produce the final output** – the MapReduce system collects all the Reduce output, and sorts it by *K2* to produce the final outcome.
- The "MapReduce Framework" orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.
- We can use such a graph to show the MapReduce algorithm.



- In our program code, we will use the **multiprocessing** library in Python3 to implement a parallel algorithm and a serial algorithm as comparasion.

Chapter 2: Data Structure / Algorithm Specification

2.1 Data Structure and Function

- We implement the MapReduce algorithm in Python3 so that the data structure in the program is some simple linear list and **dictionaries**(which is similar to the STL map in C++)
- Some important data structures are
    - pool: it is used to make a parallel algorithm and create a process pool to do the word-count parallelly. It is definited by the Python Parallel Compute Library **multiprocessing**

```
import multiprocessing
pool = multiprocessing.Pool(processes=len(file))
map_result = pool.map(word_count, file)
```

    - **path_set**: it is a list which stores all the txt files' path in the folder "data", in the program we use such an algorithm to get all the file paths

```
file_path = ".\\data\\"
    path_set = [ ]
    for file_name in os.listdir(file_path):
        path_set.append(file_path+file_name)
    return path_set
```

    - **words**: it is a line of words using file reading operation in python and we use regular expression to get rid of **all the punctuation and Escape character**(like '\n') in the text. The algorithm of those operations is as following:

```
for line in fp:
    # reduce the formats in the text.
    words = re.sub(r"[%s]+" % string.punctuation, "", line)
```

```
words = re.sub("\n", " ", words).split(' ')
```

words = [x.lower() for x in words]
for single_word in words:
    if single_word in count.keys():
        count[single_word] += 1
    else:
        count[single_word] = 1

```

    - result: the result after the **reduce operation** which merge the result of map operation.
```

```



##### 2.2 Algorithm Specification

###### 2.2.1 Overall Pseudocode

- The pseudocode of our program is

```pseudocode
Algorithm: Map-Reduce
Input: none
Output: The map of word appearance

get all the file path in the folder "data";
# the main operation of map
for i in file_path:
    single_map = word_count(i) #map the appearance of words
    add single_map to the all_maps

# the main operation of reduce
result_map = { }
for i in all_maps:
    for j in i.keys:
        result_map[j]+=1
show the result of the program
```

2.2.2 Important functions

- There are some importance functions in the program

2.2.2.1 word_count function

- We implement a parallel algorithm to count the appearance of each file.
- The function to count the appearance of the word in a single txt file, though its algorithm is simple using Python. The pseudocode of the function is:

```
Function: word_count
Input: the path of a single txt file
Return Value: a map of word appearance

open the file using the path;
while read a line from the file:
    simplify the words with reduction to punctuation and Escape character;
    change all the words into lowercase;
    split the word by ' ';
    count the appearance of each words using a dictionary.
return the dictionary as result
```

2.2.2.2 Reduce function

- The reduce function will merge the single maps into a large one using the following algorithm
    - This is a serial version, we also have a parallel version which is a little different from the following algorithm

```
Function: Reduce
Input: the single maps collected in a list
Return Value: Show the result of the total appearance

for i in map_list:
    for j in i(i is a map now):
        if j is not a digit and j is not a " ":
            result_map[j]+=i[j]
        else:
            result_map[j]=i[j]
print the result_map in alphabetical order
```

Chapter 3: Testing Results

- In this project, the test data is difficult to create by a test program. And this time, we created some text files or find some of them online to be the test data, which is in all sizes from KB to GB.
```

- What's more, we write a **test program based on the source code** to test the **running time** of our MapReduce system, we will draw a table and a statistical graph in the final of the part3.
- You can **find such test data in the folder "test data"**, but when running the code, the data will still be selected from the folder "data". In the **following parts we will not show the data in the report** but **just introduce them to avoid being a dirty report**.

3.1 Test data 1

- A single txt file with **only 1 word** in it
- It is a smallest condition of the project
- We do such a map-reduce operation 1000 times per test to calculate the average running time.
- The result is(for **100 times run**, just part of the final input)
  - In the serial program

```
Total Number of different words: 1
Total Number of different words: 1
Total Number of different words: 1
Total Number of different words: 1
Running time: 0.550546407699585
```

  - In parallel program

```
Total Number of different words: 1
Running time: 1.062551498413086
```

3.2 Test data 2

- A single txt file with only Repeated words in a medium size.
- We copy the word "apple" for 200 times to be the test data.
- The result of **100 times of running** is:
  - In the serial program

```
Total Number of different words: 2
The number of total words: 39900
Running time: 0.07810568809509277
```

  - In parallel program

```
apple 200
Total Number of different words: 2
Running time: 1.1764190196990967
```

3.3 Test data 3

- some txt files with KB-level sizes
- We do the operation 100 times per test to calculate the average running time.

类型	大小
文本文档	3 KB
文本文档	1 KB
文本文档	18 KB

- The running result of **100 times** of those files is
  - In serial program

```
Total Number of different words: 139
The number of total words: 400900
Running time: 0.9059920310974121
```

  - In parallel program

```
Total Number of different words: 139
Running time: 1.2047772407531738
```

3.4 Test data 4

- Some txt files with MB-level sizes but smaller than test data 5

类型	大小
文本文档	2 KB
文本文档	62 KB
文本文档	18 KB

- We do the operation 100 times per test to caculate the average running time.
  - In the serial program

```
Total Number of different words: 2150
The number of total words: 1591000
Running time: 3.124260902404785
```

  - In the parallel program

```
Total Number of different words: 2150
Running time: 3.4205074310302734
```

3.5 Test data 5

- some txt files with MB-level sizes

文本文档	9,583 KB
文本文档	28,749 KB
文本文档	28,722 KB

- We do the operation 10 times per test to caculate the average running time.
  - In the serial program

```
Total Number of different words: 2099
The number of total words: 132374610
Running time: 82.8415732383728
```

  - In parallel program

```
Total Number of different words: 2099
Running time: 42.72328853607178
```

3.6 Test data 6

- large amount of files with MB-level sizes(25 files in total)

大小
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB
9,583 KB

- We run this program in **one time**, the result is
  - In the serial program

```
Total Number of different words: 2097
The number of total words: 47296200
Running time: 29.439714193344116
```

  - In parallel program

```
Total Number of different words: 2097
Running time: 13.217740297317505
```

3.7 Test data 7

- A large size file with nealy 112MB data (I tried to run GB size data but my computer is out of work)
  - In the serial program

```
Total Number of different words: 2097
The number of total words: 22702192
Running time: 14.038269996643066
```

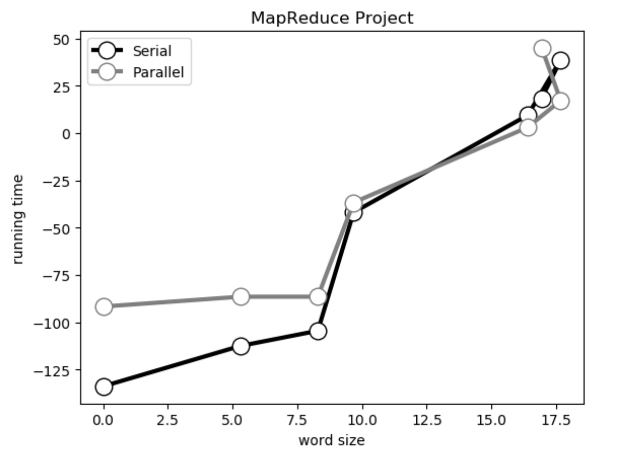
- In parallel program

```
Total Number of different words: 2097
Running time: 35.09289836883545
```

3.7 The static data

	Total Word Size	File size	running time(per time)(s) serial	running time(per time)(s) parallel
1	1	1	0.006	0.011
2	200	1	0.008	0.012
3	4000	3	0.009	0.012
4	15910	3	0.031	0.036
4	13237461	3	8.284	4.272
5	47296200	25	29.440	13.218
6	22702192	1	14.038	35.0928

- We could draw a static graph to show the result more directly
  - To make the graph a more contrast one, we use the **matplotlib** in python and do some **pre-process** to the data that is replace the origin data with its log and other operations, you can see the specific operation in our code `draw.py`
  - The analysis of the project is in the chapter 4



Chapter 4: Analysis and Comments

4.1 Analysis of Time and Space complexity

- It's a special project which is not a traditional algorithm or data structure. And it is hard to make a conclusion of its time and space complexity for the reason that the main cost depends on the total size of the words in several files.
- Actually, the topic request is to analysis the test results, which will be detailed in 4.2

4.2 Analysis of the test results

- According to the results of testing, we could have the following conclusions

- In the serial algorithm of MapReduce, the time cost has **a linear relationship** with the size of words(We assume that the size of words is M). We think the reason is that the serial algorithm will traverse all the words once and calculate the sum of their appearance, so the time complexity is $T = O(M)$ in the traditional algorithm. And the number of files doesn't make a great difference in the time cost.
- In the parallel algorithm of MapReduce, we could make a conclusion that both size of the word and file impact greatly on the time cost because in the program we will create a new process for each file to count the word appearance and finally merge them one by one. So the time complexity is $T = f(M, N)$ where N is the size of files, but it's still $O(M)$ for a certain N.
- When the test data is in a small size, the serial algorithm runs faster then the parallel algorithm. However, when the test data is in a very large size with only one file, the serial algorithm still runs faster. Only **when the size of words and files are both in a large size, the parallel algorithm will run faster than the serial algorithm**. We could infer that it's because the time cost of creating a new process for a new file to count the word appearance is large compared to the total running time. So when the size of files is just one, the parallel algorithm just waste more time than serial algorithm. But a parallel algorithm can make operations on the files together by making full use of the CPU and memory. So when the size of words and files are both suitably large, the parallel algorithm will do a good job.

Appendix

Source Code

- Source code(Parallel version)

```python
import re
import os
import time
import string
import multiprocessing

number_of_word = 0


# count the appearance of words in the text file
def word_count(file_path):
    """
    :param file_path: the path of the text file
    :return: a dictionary of words' appearance
    """
    fp = open(file_path)
    count = {}
    global number_of_word
    number = 0
    for line in fp:
        # reduce the formats in the text.
        words = re.sub(r"[%s]+" % string.punctuation, "", line)
        words = re.sub("\n", " ", words).split(' ')
        words = [x.lower() for x in words]
        # words = re.sub(r"[%s]+" % string.punctuation, "", line).split(' ')
        for single_word in words:
            number += 1
            if single_word in count.keys():
                count[single_word] += 1
            else:
                count[single_word] = 1
    # print(count)
    number_of_word += number
    # print(number)
    # print(number_of_word)
    return count


# find all the file path
def get_file_path():
    """
    :no param
    :return: a list of all the file paths
    """
```

```
    file_path = ".\\test data\\test4\\"
    path_set = []
    for file_name in os.listdir(file_path):
        path_set.append(file_path + file_name)
    return path_set


# show the result of the map-reduce algorithm
def show_result(result_map):
    """

    :param result_map: the result map that will be shown to us
    :return: none
    """
    # result_map.sort()
    count = 0
    for i in sorted(result_map.keys()):
        if count != 0:
            print(i, result_map[i])
        count += 1
    print("Total Number of different words:", count)
    # global number_of_word
    # print("The number of total words:", number_of_word)


# the main function of map operation
def Map_Operation():
    file = get_file_path()
    # print(file)
    # parallel
    pool = multiprocessing.Pool(processes=len(file))
    map_result = pool.map(word_count, file)
    # print(map_result)
    Reduce_Operation(map_result)
    # print(count_dict)


# the main function of reduce operation
def Reduce_Operation(count_dict):
    # print(count_dict)
    # print(len(count_dict))
    result = {}
    # i is a dict now
    for i in count_dict:
        # print(i)
        for j in i.keys():
            # print(j)
            if str(j).isdigit() == 0 and str(j) != " ":
                if str(j) in result.keys():
                    # print(1)
                    result[str(j)] += int(i[j])
                else:
                    # print(0)
                    result[str(j)] = int(i[j])
    # print(result)
    show_result(result)


if __name__ == '__main__':
    # test the running time of the program
    start = time.time()
    Map_Operation()
    # print(count_dict)
    # Reduce_Operation()
    end = time.time()

    print("Running time:", (end - start))

```

- Source code(Serial version)

```
import re
import os
import time
import string



count_dict = []
number_of_word = 0


# count the appearance of words in the text file
def word_count(file_path):
    """
```

```
    :param file_path: the path of the text file
    :return: a dictionary of words' appearance
    """
    fp = open(file_path)
    count = {}
    global number_of_word
    for line in fp:
        # reduce the formats in the text.
        words = re.sub(r"[%s]+" % string.punctuation, "", line)
        words = re.sub("\n", " ", words).split(' ')
        words = [x.lower() for x in words]
        # words = re.sub(r"[%s]+" % string.punctuation, "", line).split(' ')
        for single_word in words:
            number_of_word += 1
            if single_word in count.keys():
                count[single_word] += 1
            else:
                count[single_word] = 1
    # print(count)
    global count_dict
    count_dict.append(count)
    # print(count)


# find all the file path
def get_file_path():
    """
    :no param
    :return: a list of all the file paths
    """
    file_path = ".\\test data\\test4\\"
    path_set = []
    for file_name in os.listdir(file_path):
        path_set.append(file_path+file_name)
    return path_set


# show the result of the map-reduce algorithm
def show_result(result_map):
    """

    :param result_map: the result map that will be shown to us
    :return: none
    """
    # result_map.sort()
    count = 0
    for i in sorted(result_map.keys()):
        if count != 0:
            print(i, result_map[i])
        count += 1
    print("Total Number of different words:", count)


# the main function of map operation
def Map_Operation():
    file = get_file_path()
    # print(file)
    for i in file:
        word_count(i)
    # print(count_dict)


# the main function of reduce operation
def Reduce_Operation():
    global count_dict
    # print(count_dict)
    # print(len(count_dict))
    result = {}
    # i is a dict now
    for i in count_dict:
        # print(i)
        for j in i.keys():
            # print(j)
            if str(j).isdigit() == 0 and str(j) != " ":
                if str(j) in result.keys():
                    # print(1)
                    result[str(j)] += int(i[j])
                else:
                    # print(0)
                    result[str(j)] = int(i[j])
    # print(result)
    show_result(result)


if __name__ == '__main__':
    # test the running time of the program
    start = time.time()
```

```
    Map_Operation()
    # print(count_dict)
    Reduce_Operation()
    end = time.time()
    print("The number of total words:", number_of_word)
    print("Running time:", end-start)
```

- Test Program is adapted from the source code.

References

List all the references here in the following format:

[1] 《算法导论》第三版

[2] PTA website https://pintia.cn/

[3] Course Slides of ADS

Author List

- Zhang Yichi 3180103772
- Zhang Qi 3180103162
- Nie Junzhe 3180103501

Declaration

We hereby declare that all the work done in this project titled "Huffman Codes" is of our independent effort as a group.

Signatures