Project 7
MapReduce

Meng JunYi

Date: June 20, 2021

Chapter 1 : Introduction

1 Background

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a Map() procedure and a Reduce() procedure.

2 Problem Description

In this project, you are supposed to briefly introduce the framework of MapReduce (How does it work?), and implement a MapReduce program to count the number of occurrences of each word in a set of documents.

3 Output Specification

Your task includes the following steps:

1. Setup MapReduce libraries. A popular open-source implementation is Apache Hadoop.
2. Write a parallel MapReduce program and a serial program to solve this problem. You are supposed to print your results in non-increasing order of the number of occurrences of words. If two or more words have the same number of occurrences, they must be printed in lexicographical order. Make sure that each line contains one word, followed by its number of occurrences, separated by a space, and there must be no extra space at the end of each line.
3. Prepare appropriate test data. A set of documents (files) which contains a minimum of 100,000 words must be used for testing.
4. Test your programs; make sure that the results are accurate. Then compare and analyze the performances between parallel and serial algorithms.

Chapter 2: Algorithm

Algorithm: Serial Algorithm

The idea is quite simple, just continuously read one word and change result until the file reaches end.

```
Init(res)
for each word in file:

       res[word] ++

sort(res)
for each word in res.keys():
       output(word)
       output(res[word])
```

Algorithm: MapReduce

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 is composed of a map procedure, which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a reduce method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "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.

In implementation, we firstly separate input file into parts, then for each part, carry out Serial Algorithm, store its result (Map) and finally, sum them up to obtain the final result (Reduce).

```
// Map procedure
Init(res)
Break text into parts
for each part in parts:
       for each word in part:

              res[part][word] ++
// Reduce procedure
Init(final_res)
for each part in parts:
       for each word in part:
              final_res[word] += res[part][word]
sort and output final_res
```

Chapter 3: Testing Results

In this project, I made both input and output through file IO, to avoid the time measurement error that "output to screen" process brings, since the scales of both input and output can be quite large.

In all our test cases, both the serial algorithm and the parallel algorithm give the same answer, so it is sure that all programs are correct. And efficiency will be discussed in next chapter.

Case 1

Content of input.txt	**Content of output.txt**
i i i hello hello hello hello hello hello hello hello world world world world world world world world world	world 9 hello 8 i 3

Case 2

Content of input.txt	**Content of output.txt**
a a a b b b c c c ca ca ca	a 3 b 3 c 3 ca 3

Case 3

In this final case, I downloaded an E-book from Internet and use this natural text to verify its correctness.
To do with natural text, I remove all punctuations in the text and convert all upper case chars into lower case.

You can check this file and its output result in ***code*** folder.

Chapter 4: Analysis and Comments

Comparison

First of all, in my implementation, I found that map of C++ stl is quite slow than building a Trie to count the words. So I wrote another version that uses Trie to count words and the parallel algorithm also use Trie to count the words for each part of the text.

For the natural text:

Serial Using STL	0.726000 seconds
Serial Using Trie	0.236000 seconds
Parallel Using Trie	0.132000 seconds

My parallel program use 8 maps to count the words at the same time, larger number of maps don't help much in reducing time.

Analysis

Suppose we have ***M*** kinds of words with a total count *of **N**,* the longest word in the text contains ***MAX*** characters, and use ***K*** maps.

Time complexity

Since the algorithm that uses STL is quite related to its implement that is unknown to users, we don't discuss about it here.

Serial Using Trie

The worst time complexity for each insertion into Tries is $O(MAX)$, so $O(N * MAX)$ is needed for counting the words, then $O(M * logM)$ to sort the result.

The total time complexity is $O(N * MAX + M * logM)$

Parallel Using Trie

The worst time complexity for each insertion into Tries is $O(MAX)$, so $O(N * MAX)$ is needed for counting the words, then $O(M * logM)$ to sort the result.

Besides, the reduce procedure needs extra time to sum up the result, this work takes $O(K * N)$ (in worst case, each word appears in each part of the test).

The total time complexity is $O(N * MAX + M * logM + K * N)$

Space complexity

Serial Using Trie

$O(N)$, since there is only one tree that actually build.

Parallel Using Trie

$O(K * N)$, in worst case, each word appears in each part of the test, so for every part of the text, a full-sized tree needs to be build.

Comments:

Although the workload seems to be larger for parallel algorithm, but due to its full use of the advantages of multi-core, parallel algorithm works much better for large scale data.

Declaration

I hereby declare that all the work done in this project titled "Map Reduce" is of my independent effort.

Appendix: Source Code

Serial Using STL

```
#include <iostream>
#include <sstream>
#include <map>
#include <vector>
#include <ctime>
#include <cstring>
#include <algorithm>

using namespace std;

// time related
clock_t start, finish;
double duration;
// word related
char word [1000];
int loc;

bool cmp(const pair<string, int>& a, const pair<string, int>& b) {
        if (a.second != b.second)
```

```
            return a.second > b.second;
        else
            return a.first < b.first;
}

int main()
{
    start = clock();

    FILE * fp  = fopen("input.txt","rb");
    fseek(fp, 0, SEEK_END);
    int length  = ftell(fp);
    char * buf = (char *)malloc(length+1);
    rewind(fp);
    fread(buf, 1, length, fp);
    fclose(fp);

    buf[length] = 0;
    stringstream ss(buf);
    map<string, int> words;
    while(ss >> word){
        loc = 0;
        char * s = (char *)malloc(strlen(word)+1);
        for (int count = 0; word[count]; count++){
            if(word[count] <= 'z' && word[count] >= 'a') s[loc++] = word[count];
            if(word[count] <= 'Z' && word[count] >= 'A') s[loc++] = word[count] + 32;
        }
        if(!loc){free(s);continue;};
        s[loc] = 0;
        words[string(s)]++;
    }
    // Sort
    vector<pair<string, int>> vec(words.begin(), words.end());
    sort(vec.begin(), vec.end(), cmp);

    // Write to file
    FILE * out  = fopen("output.txt","w");
    for (int i = 0; i < vec.size(); ++i)
        fprintf(fp,"%s %d\n",vec[i].first.c_str(), vec[i].second);
    fclose(out);

    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf( "%f seconds\n", duration);
    return 0;
}
```

Serial Using Trie

```
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <cstring>

#include <iostream>
#include <vector>
#include <sstream>
#include <algorithm>
using namespace std;

// time related
clock_t start, finish;
```

```
double duration;

// word related
int loc;
char word [1000];

// Compare
bool cmp(const pair<string, int>& a, const pair<string, int>& b){
        if (a.second != b.second)
            return a.second > b.second;
        else
            return a.first < b.first;
}

// data structure related
typedef struct tree
{
    struct tree * next [26];
    char * v;
    int freq;
}* Tree;



void traverse(Tree t, vector<pair<char *, int>> & vec){
    if(t->v) vec.push_back(pair<char *, int>(t->v, t->freq));
    for (int i = 0; i < 26; i++)
    {
        if(t->next[i]) traverse(t->next[i], vec);
    }
}

Tree insert(Tree t, char * v, int start){
    if(!t){
        t = (Tree)malloc(sizeof(struct tree));
        memset(t, 0, sizeof(struct tree));
    }
    // Next character is '/0'
    if(!v[start+1]){
        if(t -> freq == 0)
            t -> v = v;
        else
            free(v);
        t -> freq++;
    }else{
        t->next[v[start+1] - 'a'] = insert(t->next[v[start+1] - 'a'], v, start+1);
    }
    return t;
}

// entry
int main(void)
{
    start = clock();

    FILE * fp  = fopen("input.txt","rb");
    fseek(fp, 0, SEEK_END);
    int length  = ftell(fp);
    char * buf = (char *)malloc(length+1);
    rewind(fp);
    fread(buf, 1, length, fp);
    fclose(fp);

    Tree dummy = (Tree)malloc(sizeof(struct tree));
    memset(dummy, 0, sizeof(struct tree));
    buf[length] = 0;
    stringstream ss(buf);
    while(ss >> word){
        loc = 0;
        char * s = (char *)malloc(strlen(word)+1);
        for (int count = 0; word[count]; count++){
            if(word[count] <= 'z' && word[count] >= 'a') s[loc++] = word[count];
            if(word[count] <= 'Z' && word[count] >= 'A') s[loc++] = word[count] + 32;
        }
        if(!loc){free(s);continue;};
        s[loc] = 0;
        dummy->next[s[0] - 'a'] = insert(dummy->next[s[0] - 'a'], s, 0);
    }

    // Sort
    vector<pair<char *, int>> vec;
    vec.reserve(30000);
    traverse(dummy, vec);
    sort(vec.begin(), vec.end(), cmp);
    // Write to file
    FILE * out  = fopen("output.txt","w");
    for (int i = 0; i < vec.size(); ++i)
        fprintf(fp,"%s %d\n",vec[i].first, vec[i].second);
    fclose(out);

    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf( "%f seconds\n", duration);
    return 0;
}
```

Parallel Using Trie

```
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <cstring>

#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <algorithm>
#include <string>
#include <pthread.h>

using namespace std;

// multithreading related
#define CORES 4
#define HYPERTHREADING 2
#define NUM_THREADS CORES*HYPERTHREADING

// time related
clock_t start_time, finish_time;
double duration;

// Compare
bool cmp(const pair<string, int>& a, const pair<string, int>& b){
        if (a.second != b.second)
            return a.second > b.second;
        else
            return a.first < b.first;
}
```

```
// data structure related
typedef struct tree
{
    struct tree * next [26];
    char * v;
    int freq;
}* Tree;

void traverse(Tree t, map <string, int> & words){
    if(t->v){
        string * v = new string(t->v);
        words[*v] =  t->freq;
    }
    for (int i = 0; i < 26; i++)
    {
        if(t->next[i]) traverse(t->next[i], words);
    }
}

Tree insert(Tree t, char * v, int start){
    if(!t){
        t = (Tree)malloc(sizeof(struct tree));
        memset(t, 0, sizeof(struct tree));
    }
    // Next character is '/0'
    if(!v[start+1]){
        if(t -> freq == 0)
            t -> v = v;
        else
            free(v);
        t -> freq++;
    }else{
        t->next[v[start+1] - 'a'] = insert(t->next[v[start+1] - 'a'], v, start+1);
    }
    return t;
}

// Thread related
void * One_block(void * args){
    int loc;
    char word [1000];
    char * buf = (char *) args;
    stringstream ss(buf);
    Tree dummy = (Tree)malloc(sizeof(struct tree));
    memset(dummy, 0, sizeof(struct tree));
    map <string, int> * words =  new map<string, int>;
    while(ss >> word){
        loc = 0;
        char * s = (char *)malloc(strlen(word)+1);
        for (int count = 0; word[count]; count++){
            if(word[count] <= 'z' && word[count] >= 'a') s[loc++] = word[count];
            if(word[count] <= 'Z' && word[count] >= 'A') s[loc++] = word[count] + 32;
        }
        if(!loc){
            free(s);
            continue;
        };
        s[loc] = 0;
        dummy->next[s[0] - 'a'] = insert(dummy->next[s[0] - 'a'], s, 0);
    }
    traverse(dummy, * words);
    return (void*)words;
}
```

```
// entry
int main(void)
{
    start_time = clock();

    // Read from file
    FILE * fp  = fopen("input.txt","rb");
    fseek(fp, 0, SEEK_END);
    int length  = ftell(fp);
    char * buf = (char *)malloc(length+1);
    rewind(fp);
    fread(buf, 1, length, fp);
    fclose(fp);
    buf[length] = 0;

    // Separate into blocks
    pthread_t tids[NUM_THREADS];

    int block_length = length/(NUM_THREADS);
    map<string, int> * words_per_block [NUM_THREADS];
    int start = 0, end = block_length;
    for (int i = 0; i < NUM_THREADS; i++){
        while (buf[end] != ' ' && buf[end] != 0 && buf[end] != '\n' && buf[end] !='\r')
 end++;
        buf[end] = 0;
        pthread_create(&tids[i], NULL, One_block, buf+start);
        start = end + 1;
        end = min(start + block_length, length);
    }

    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(tids[i], (void **)(words_per_block+i));
    }

    // Merge
    map <string, int> words;
    map <string, int>::iterator iter;
    for (int i = 0; i < NUM_THREADS; i++){
        for(iter = words_per_block[i]->begin(); iter != words_per_block[i]-
>end(); iter++){
            words[iter->first] += iter->second;
        }
    }
    // Sort
    vector<pair<string, int>> vec(words.begin(), words.end());
    sort(vec.begin(), vec.end(), cmp);

    // Write to file
    FILE * out  = fopen("output.txt","w");
    for (int i = 0; i < vec.size(); ++i)
        fprintf(fp,"%s %d\n",vec[i].first.c_str(), vec[i].second);
    fclose(out);

    finish_time = clock();
    duration = (double)(finish_time - start_time) / CLOCKS_PER_SEC;
    printf( "%f seconds\n", duration);
    return 0;
}
```