并行分布式计算期末 Project

计算机科学技术学院　　周训哲 20307110315

2023 年 5 月 31 日

多线程并行快速排序算法

本实验的实现过程主要是使用 C++ 版本的 openMP 编程实现并行快速排序算法。在测试加速比的过程当中使用了 1000，5000，10000，100000，1000000，10000000，100000000 的数据量，发现随着数据量的增加，并行计算的加速比越大，符合预期。

一、模型介绍

openMP 介绍

OpenMP（Open Multi-Processing）是一种并行编程模型，用于在共享内存系统中编写多线程并行程序。它提供了一组指令和编译器指导，可以将串行程序转换为并行程序，以充分利用多核处理器和共享内存体系结构的性能。

OpenMP 的编程模型基于共享内存的概念，其中多个线程可以并行执行任务。在 OpenMP 中，程序员使用指令和编译器指导来标识可并行化的代码段，并指定线程之间的同步和数据共享方式。这样，程序员可以将任务分解成多个并行的线程，每个线程负责处理任务的一个部分。

快速排序介绍

快速排序（Quick Sort）是一种常用的排序算法，它基于分治（Divide and Conquer）的策略来进行排序。快速排序的核心思想是通过选择一个基

准元素，将待排序的序列分割成两个子序列，使得左侧子序列中的所有元素都小于等于基准元素，右侧子序列中的所有元素都大于等于基准元素，然后对两个子序列递归地应用快速排序算法，直到序列变为有序。

快速排序是一种天然适合并行化的排序算法，因为它的分治策略可以很容易地将排序任务分解成多个子任务，每个子任务独立地对子序列进行排序。这使得快速排序可以通过并行计算来加速排序过程，提高排序的效率。

在并行快速排序中，可以使用多线程或并行计算模型来同时处理多个子任务。每个线程或计算单元可以负责处理一个子序列，使用快速排序算法进行排序。然后，将子序列合并成最终的有序序列。

常见的并行快速排序算法的步骤：

1. 并行划分：选择一个基准元素，并使用并行计算或多线程来划分序列，使得每个线程或计算单元独立地处理子序列的一部分。每个线程或计算单元将小于等于基准元素的元素放在左侧，大于等于基准元素的元素放在右侧。

2. 并行排序：对左侧和右侧的子序列分别应用并行快速排序算法，每个线程或计算单元递归地对子序列进行排序。

3. 合并结果：等待所有线程或计算单元完成排序后，将各个子序列按照顺序合并，得到最终的有序序列。

二、模型实现

代码主要使用 C++ 进行实现，调用 <omp.h> 中的库函数，使用 #pragma omp 语句实现并行计算。

在实现 openMP 完成快速排序的过程中，主要将代码划分成 quickSort 和 main 两个部分，quickSort 定义了串行和并行算法的计算过程，主函数则调用算法对串行和并行算法的加速比进行计算。

同时为了实现生成固定长度的数据集以及一个串行算法用于测算对应的加速比，

模型代码实现如下

```
#include "omp.h"
#include <stdlib.h>
```

```
3     //随机创建数组
4     void rands(int* data, int sum);
5     //交换函数
6     void swap(int* a, int* b);
7     //求2的n次幂
8     int exp2(int wht_num);
9     //求log2(n)
10    int log2(int wht_num);
11    //合并两个有序的数组
12    void mergeList(int* c, int* a, int sta1, int end1, int* b,
      int sta2, int end2);
13    //串行快速排序
14    int partition(int* a, int sta, int end);
15    void quickSort(int* a, int sta, int end);
16    //openMP(8)并行快速排序
17    void quickSort_parallel(int* array, int lenArray, int
      numThreads);
18    void quickSort_parallel_internal(int* array, int left, int
      right, int cutoff);
19
20    void rands(int* data, int sum) {
21      int i;
22      for (i = 0; i < sum; i++)
23      {
24        data[i] = rand() % 1000000000;
25      }
26    }
27
28    void swap(int* a, int* b) {
29      int t = *a;
30      *a = *b;
31      *b = t;
32    }
33    int exp2(int wht_num) {
34      int wht_i;
35      wht_i = 1;
36      while (wht_num > 0)
37      {
```

```
38          wht_num--;
39          wht_i = wht_i * 2;
40        }
41        return wht_i;
42      }
43      int log2(int wht_num) {
44        int wht_i, wht_j;
45        wht_i = 1;
46        wht_j = 2;
47        while (wht_j < wht_num)
48        {
49          wht_j = wht_j * 2;
50          wht_i++;
51        }
52        if (wht_j > wht_num)
53        wht_i--;
54        return wht_i;
55      }
56      int partition(int* a, int sta, int end) {
57        int i = sta, j = end + 1;
58        int x = a[sta];
59        while (1)
60        {
61          while (a[++i] < x && i < end);
62          while (a[--j] > x);
63          if (i >= j)
64          break;
65          swap(&a[i], &a[j]);
66        }
67        a[sta] = a[j];
68        a[j] = x;
69        return j;
70      }
71      //并行openMP排序
72      void quickSort_parallel(int* a, int lenArray, int numThreads
        ) {
73        int cutoff = 100;
74        #pragma omp parallel num_threads(numThreads) //指定线程数
```

```
    的数量
75      {
76        #pragma omp single //串行执行
77        {
78          quickSort_parallel_internal(a, 0, lenArray - 1, cutoff
    );
79        }
80      }
81    }
82    void quickSort_parallel_internal(int* a, int left, int right
    , int cutoff) {
83      int i = left, j = right;
84      int tmp;
85      int mid = a[(left + right) / 2];
86      //进行数组分割, 分成两部分（符合左小右大）
87      while (i <= j)
88      {
89        while (a[i] < mid)
90        i++;
91        while (a[j] > mid)
92        j--;
93        if (i <= j) {
94          tmp = a[i];
95          a[i] = a[j];
96          a[j] = tmp;
97          i++;
98          j--;
99        }
100     }
101     //int j = partition(a,left,right);
102     if (((right - left) < cutoff)) {
103       if (left < j) {
104         quickSort_parallel_internal(a, left, j, cutoff);
105       }
106       if (i < right) {
107         quickSort_parallel_internal(a, i, right, cutoff);
108       }
109     }
```

```
110       else {
111         #pragma omp task
112         //对两部分再进行并行的线程排序
113         {
114           quickSort_parallel_internal(a, left, j, cutoff);
115         }
116         #pragma omp task
117         {
118           quickSort_parallel_internal(a, i, right, cutoff);
119         }
120       }
121     }
122     //合并两个已排序的数组
123     void mergeList(int* c, int* a, int sta1, int end1, int* b,
    int sta2, int end2) {
124       int a_index = sta1; // 遍历数组a的下标
125       int b_index = sta2; // 遍历数组b的下标
126       int i = 0;          // 记录当前存储位置
127       while (a_index < end1 && b_index < end2) {
128         if (a[a_index] <= b[b_index]) {
129           c[i] = a[a_index];
130           a_index++;
131         }
132         else {
133           c[i] = b[b_index];
134           b_index++;
135         }
136         i++;
137       }
138       while (a_index < end1) {
139         c[i] = a[a_index];
140         i++;
141         a_index++;
142       }
143       while (b_index < end2) {
144         c[i] = b[b_index];
145         i++;
146         b_index++;
```

```
147       }
148     }
149     //串行快速排序
150     void quickSort(int* a, int sta, int end) {
151       if (sta < end) {
152         //printf("3\n");
153         int mid = partition(a, sta, end);
154         quickSort(a, sta, mid - 1);
155         quickSort(a, mid + 1, end);
156       }
157     }
158
```

主函数

```
1   #include <stdio.h>
2   #include <omp.h>
3   #include <stdlib.h>
4   #include <time.h>
5   #include "quickSort.h"
6
7   void test(int n) {
8     omp_set_num_threads(8);
9     double omp_time_begin, omp_time_end;
10    double seq_time_begin, seq_time_end;
11    double omp_time, seq_time;
12    int* data1, * data2;
13    int num = 8;
14    data1 = (int*)malloc(sizeof(int) * n);
15    data2 = (int*)malloc(sizeof(int) * n);
16    rands(data1, n);
17    rands(data2, n);
18
19    //并行快速排序
20    omp_time_begin = omp_get_wtime();
21    quickSort_parallel(data1, n, num);
22    omp_time_end = omp_get_wtime();
23    omp_time = omp_time_end - omp_time_begin;
24
```

```
25        //串行快速排序
26        seq_time_begin = omp_get_wtime();
27        quickSort(data2, 0, n - 1);
28        seq_time_end = omp_get_wtime();
29        seq_time = seq_time_end - seq_time_begin;
30
31        //输出运行时间
32        printf("————————————————————\n");
33        printf("测试数据量 : %d\n", n);
34        printf("并行处理时间 : %lf s\n", omp_time);
35        printf("串行处理时间 : %lf s\n", seq_time);
36        printf("加速比 : %lf\n", seq_time / omp_time);
37        printf("————————————————————\n");
38        printf("\n");
39
40        //释放内存
41        free(data1);
42        free(data2);
43    }
44
45    int main()
46    {
47        test(1000);
48        test(5000);
49        test(10000);
50        test(100000);
51        test(1000000);
52        test(10000000);
53        test(100000000);
54        return 0;
55    }
56
```

三、测试结果

测试数据量	1000	5000	10000	100000
并行处理时间	0.000206 s	0.000632 s	0.001312 s	0.014380 s
串行处理时间	0.000256 s	0.000856 s	0.001788 s	0.021190 s
加速比	1.241881	1.355978	1.363345	1.473619

测试数据量	1000000	10000000	100000000
并行处理时间	0.150549 s	1.477709 s	9.344500 s
串行处理时间	0.241012 s	2.706674 s	18.478509 s
加速比	1.600890	1.831669	1.977474

MapReduce 实现 wordcount

本实验有两种实现方式，其中一种是使用 openMP 并行框架实现（使用 C++ 实现）；另外一种是使用 hadoop 实现（使用 python 实现）。在测试加速比过程中为了对比 hadoop 的加速的提升，使用 python 计算了 1K、1M、10M、100M 数据量下的加速比。

一、模型介绍

hadoop 介绍

Hadoop 是一个开源的分布式计算框架，旨在处理大规模数据集的存储和处理。它提供了可靠性、可扩展性和容错性，使得它成为处理大规模数据的理想选择。

Hadoop 的核心组件包括以下几个部分：

1.Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）：HDFS 是 Hadoop 的分布式文件系统，用于存储大规模数据集。它具有高容错性和可靠性，通过将数据分布在集群中的多个节点上实现数据冗余和容错。

图 1: openMP 快排加速比

2.Hadoop YARN（Yet Another Resource Negotiator）：YARN 是 Hadoop 的资源管理器，负责集群资源的管理和作业调度。它允许用户在 Hadoop 集群上运行各种计算框架，如 MapReduce、Spark 和 Flink 等。

3.MapReduce：MapReduce 是 Hadoop 的计算模型和编程范式。它将大规模数据集分割为小的数据块，并将其分发到集群中的多个计算节点上进行并行处理。MapReduce 提供了数据处理的分布式框架，包括数据划分、并行计算、数据排序和结果汇总等功能。

MapReduce 介绍

MapReduce 是一种分布式计算模型和编程范式，最初由 Google 提出，并成为 Hadoop 中的核心组件之一。它旨在解决大规模数据集的并行处理问题。

MapReduce 模型基于两个主要操作：Map 操作和 Reduce 操作。下面是对 MapReduce 模型的简要说明：

1.Map 操作：Map 操作是将输入数据集中的每个元素映射到一组键-值对（Key-Value Pair）。Map 操作通常会并行处理数据集的各个部分，将输入数据划分为多个片段，然后每个片段由一个 Map 任务处理。Map 任务对每个输入元素应用一组操作，生成键-值对作为中间结果。

2.Shuffle 和排序：在 Map 操作之后，框架会自动执行 Shuffle 和排序操作。Shuffle 的目的是将 Map 任务输出的键-值对重新分配到不同的 Reduce 任务上。在 Shuffle 过程中，相同键的键-值对会被分组并发送到同一个 Reduce 任务进行处理。此外，Shuffle 还会对键进行排序，以便在 Reduce 阶段进行更高效的处理。

3.Reduce 操作：Reduce 操作接收 Shuffle 和排序阶段的输出作为输入，并根据键对值进行聚合、汇总或其他计算。Reduce 任务会针对每个唯一的键处理一组值，生成最终的输出结果。Reduce 操作通常也是并行处理的，多个 Reduce 任务可以同时处理不同的键集。

二、模型实现

代码在 openMP 过程中主要使用 C++ 进行实现，调用 <omp.h> 中的库函数，使用 #pragma omp 语句实现并行计算。

在使用 hadoop 过程中主要使用 Python 进行实现，分别编写 mapper 和 reducer 进行 map 和 reduce 的操作。

在实现 openMP 完成快速排序的过程中，主要将代码划分成 quickSort 和 main 两个部分，quickSort 定义了串行和并行算法的计算过程，主函数则调用算法对串行和并行算法的加速比进行计算。

模型代码实现如下

openMP 实现 MapReduce 下的 wordcount

```
#include<stdc++.h>
#include<omp.h>
using namespace std;
#define NUMPROCS 8
#include <iostream>
#include <string>
#include <vector>
#include <io.h>
using namespace std;


//核心代码
void getFileNames(string path, vector<string>& files)
{
  //文件句柄
  intptr_t hFile = 0;
  //文件信息
  struct _finddata_t fileinfo;
  string p;
  if ((hFile = _findfirst(p.assign(path).append("/*").c_str
(), &fileinfo)) != -1)
  {
    do
    {
      //如果是目录,递归查找
      //如果不是,把文件绝对路径存入vector中
      if ((fileinfo.attrib & _A_SUBDIR))
      {
        if (strcmp(fileinfo.name, ".") != 0 && strcmp(
```

```
    fileinfo.name, "..") != 0)
29              getFileNames(p.assign(path).append("/").append(
    fileinfo.name), files);
30          }
31          else
32          {
33              files.push_back(p.assign(path).append("/").append(
    fileinfo.name));
34          }
35      } while (_findnext(hFile, &fileinfo) == 0);
36      _findclose(hFile);
37    }
38  }
39
40
41  struct word_reader : std::ctype<char> {
42    word_reader(std::string const& delims) : std::ctype<char>(
    get_table(delims)) {}
43    static std::ctype_base::mask const* get_table(std::string
    const& delims) {
44      static std::vector<std::ctype_base::mask> rc(table_size,
     std::ctype_base::mask());
45
46      for (char ch : delims)
47      rc[ch] = std::ctype_base::space;
48      return &rc[0];
49    }
50  };
51
52  int main() {
53    omp_set_num_threads(NUMPROCS);
54    int num = NUMPROCS;
55    string str;
56    vector<string> vals;
57    vector<string> fileNames;
58
59    string path("C:/Users/ZhouXunZhe/Desktop/WordCount/input")
    ;   //自己选择目录测试
```

```
60      getFileNames(path, fileNames);
61      for (const auto& file : fileNames) {
62        ifstream inpstream;
63        inpstream.open(file);
64        while (getline(inpstream, str)) {
65          istringstream in(str);
66          in.imbue(std::locale(std::locale(), new word_reader(",.\r\n")));
67          std::string word;
68
69          while (in >> word) {
70            //std::cout << word << "\n";
71            vals.push_back(word);
72          }
73        }
74      }
75      int len = vals.size();
76      string* arr = new string[len];
77      double starttime = omp_get_wtime();
78      for (int i = 0; i < len; i++) {
79        arr[i] = vals[i];
80      }
81      map<string, int> mapper[NUMPROCS];
82      #pragma omp parallel for
83      for (int i = 0; i < len; i++)
84      {
85        if (mapper[omp_get_thread_num()].find(arr[i]) != mapper[omp_get_thread_num()].end()) {
86          mapper[omp_get_thread_num()][arr[i]] = 1;
87
88        }
89        else
90        {
91          mapper[omp_get_thread_num()][arr[i]] += 1;
92        }
93
94      }
95      map <string, int> res;
```

```
96      #pragma omp critical
97      {   for (int i = 0; i < num; i++)
98        for (map<string, int>::iterator it = mapper[i].begin();
    it != mapper[i].end(); it++) {
99          if (res.find(it->first) != res.end())
100         res[it->first] += it->second;
101         else
102         res[it->first] = it->second;
103       }
104     }
105     cout << omp_get_wtime() - starttime << endl;
106     FILE* ptr2 = fopen("C:/Users/ZhouXunZhe/Desktop/WordCount/
    resultopenmp.dat", "a");
107     fprintf(ptr2, "%d\t%f\n", num, omp_get_wtime() - starttime
    );
108     fclose(ptr2);
109     FILE* fptr = fopen("C:/Users/ZhouXunZhe/Desktop/WordCount/
    output/outputopenmp.txt", "w+");
110     for (map<string, int>::iterator it = res.begin(); it !=
    res.end(); it++)
111     fprintf(fptr, "%s\t%d\n", (it->first).c_str(), it->second)
    ;
112   }
113
```

python 实现随机段落生成

```
1 import random
2 from nltk.corpus import words
3
4 MAX_WORDS = 5
5 word_list = words.words()
6 my_word_list = {}
7 for i in range(1, MAX_WORDS+1):
8         filtered_words = [s for s in word_list if len(s) == i]
9         my_word_list[i] = filtered_words
```

```
10
11
12 def generate_word(length):
13         filtered_strings = my_word_list[length]
14         if filtered_strings:
15                 return random.choice(filtered_strings)
16         else:
17                 return None
18
19
20 def generate_sentence(length):
21         sentence = []
22         while len(sentence) < length:
23                 word_len = random.randint(1, MAX_WORDS)
24                 word = generate_word(word_len)
25                 sentence.append(word)
26         return ' '.join(sentence)
27
28
29 def generate_article(length):
30         article = []
31         word_size = 0
32         while word_size < length:
33                 sentence_length = random.randint(5, 15)
34                 sentence = generate_sentence(sentence_length)
35                 article.append(sentence)
36                 word_size += sentence_length
37         return '\n'.join(article)
38
39
40 length = 1000
```

```
article = generate_article(length)
with open('C:/Users/ZhouXunZhe/Desktop/WordCount/input/' + str(length) + '.txt',
        f.write(article)
```

python 实现串行 wordcount

```

import time
import os
import operator

input_path = 'C:/Users/ZhouXunZhe/Desktop/WordCount/input/'
wordcount = {}
start = time.perf_counter()

for root, dirs, files in os.walk(input_path):
        for file in files:
                with open(input_path + file, 'r') as f:
                        lines = f.readlines()
                        for line in lines:
                                words = line.split()
                                for word in words:
                                        if word not in wordcount:
                                                wordcount[word] = 0
                                        wordcount[word] += 1
wordcount = dict(sorted(wordcount.items(), key=operator.itemgetter(0)))
with open('C:/Users/ZhouXunZhe/Desktop/WordCount/output/myoutput.txt', 'w') as g
        for word in wordcount.keys():
                g.write('%s\t%s\n' % (word, wordcount[word]))

end = time.perf_counter()
print(str(end - start) + ' s')
```

python 使用 hadoop 实现 wordcount

```
# mapper.py
import sys

for line in sys.stdin:
        line = line.strip()
        words = line.split()
for word in words:
        print("%s\t%s" % (word, 1))

# reducer.py
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
        line = line.strip()
        word, count = line.split('\t', 1)
        try:
                count = int(count)
        except ValueError:
                continue
        if current_word == word:
                current_count += count
        else:
                if current_word:
                        print("%s\t%s" % (current_word, current_count))
```

```
29                  current_count = count
30                  current_word = word
31
32 if word == current_word:
33         print("%s\t%s" % (current_word, current_count))
```

三、测试结果

最终测试时采用单个线程与 8 个线程的 mapreduce 算法进行比较计算加速比。

测试数据量	1000	1M	10M	100M
串行处理时间	0.0147531 s	12.2467 s	170.093 s	3326.44 s
并行处理时间	0.0145879 s	8.33072 s	80.9971 s	786.0211 s
加速比	1.0113	1.4701	2.0981	4.2319

可以看到随着数据量的提升，加速比也有十分大的提升。

python 程序中测试使用 Hadoop 的结果对于长度 1M 的数据需要运行 9s，而使用 python 中的 map 函数则只需要 0.3s

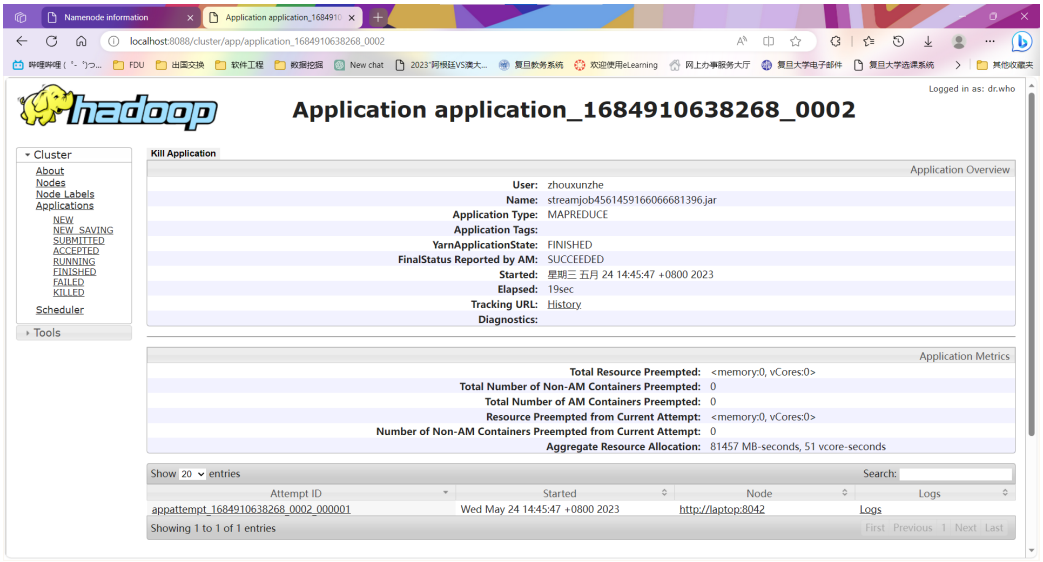

图 2: 使用 hadoop 进行 mapreduce 计算

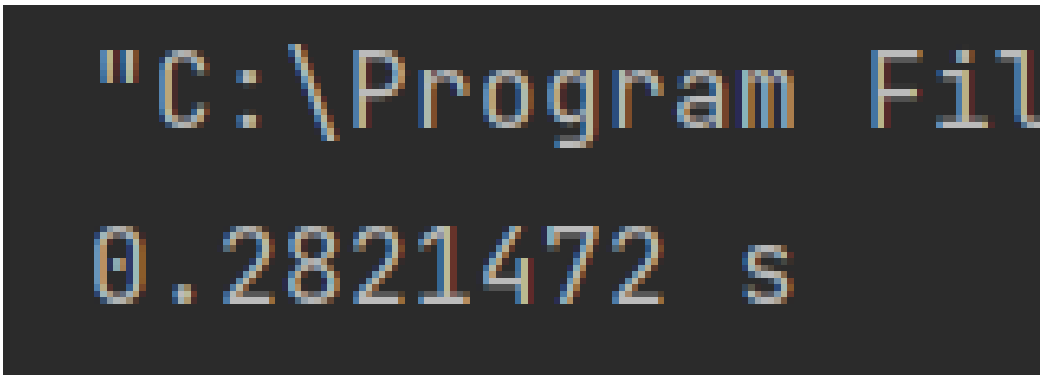

图 3: 使用 python 中的 map 库进行 wordcount

实验总结

在本次实验中，我第一次接触到并行计算的代码实现。在本节课程以及本次课程 project 中，我学会了很多并行编程的技巧和掌握了并行编程的能力。除此以外，我觉得最为关键的就是我自己搭建了 Hadoop 的计算环境。该项工作为我之后的大数据处理奠定了一定的基础。通过本次 pj，我更加深刻理解了 openMP 和 MapReduce 的编程原理以及并行计算对于大数据计算的重要性。

非常感谢老师以及助教对本次项目的设计以及问题的解答。