并行与分布式系统期末作业

TSP 问题并行化求解

19335015 陈恩婷

2020-1-22

目录

1. 概述

1. 问题与解法描述

旅行商问题（英语：Travelling salesman problem, TSP）是组合优化中的一个 NP 困难问题，在运筹学和理论计算机科学中非常重要。问题内容为“给定一系列城市和每对城市之间的距离，求解访问每一座城市一次并回到起始城市的最短回路。”

深度优先搜索算法（英语：Depth-First-Search，DFS）是一种用于遍历或搜索树或图的算法。这个算法会尽可能深的搜索树的分支。当节点 v 的所在边都已经搜索过时，搜索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。

2. 实验内容

本实验根据《并行程序设计导论》(An introduction to parallel programming) 中的介绍，分别实现了两种串行程序、OpenMP、Pthreads 的深度优先搜索的 TSP 问题求解。

2. 实验过程

1. 串行版本

串行的 DFS 实现方法共有三种，第一种是递归调用的解法：

```
1  void Depth_first_search(tour_t tour) {
2     city_t city;
3
4     if (City_count(tour) == n) {
5        if (Best_tour(tour))
6           Update_best_tour(tour);
7     } else {
8        for each neighboring city
9           if (Feasible(tour, city)) {
10             Add_city(tour, city);
11             Depth_first_search(tour);
12             Remove_last_city(tour, city);
13          }
14    }
15 }  /* Depth_first_search */
```

Program 6.4: Pseudocode for a recursive solution to TSP using depth-first search

第二种是使用栈的迭代解法，其中栈中存储的是接下来要搜索的城市（一次推入或弹出一个城市）：

```
1  for (city = n-1; city >= 1; city--)
2     Push(stack, city);
3  while (!Empty(stack)) {
4     city = Pop(stack);
5     if (city == NO_CITY) // End of child list, back up
6        Remove_last_city(curr_tour);
7     else {
8        Add_city(curr_tour, city);
9        if (City_count(curr_tour) == n) {
10          if (Best_tour(curr_tour))
11             Update_best_tour(curr_tour);
12          Remove_last_city(curr_tour);
13       } else {
14          Push(stack, NO_CITY);
15          for (nbr = n-1; nbr >= 1; nbr--)
16             if (Feasible(curr_tour, nbr))
17                Push(stack, nbr);
18       }
19    }  /* if Feasible */
20 }  /* while !Empty */
```

Program 6.5: Pseudocode for an implementation of a depth-first solution to TSP that doesn't use recursion

实现搜索逻辑的函数代码如下：

```
Tour tsp() {
    stack<city> s;
    s.empty();
    Tour curr_tour, best;
    curr_tour.cost = 0;
    curr_tour.route.clear();
    curr_tour.route.push_back( 0 );
    best.cost = INT_MAX;
    best.route.clear();
    for ( city i = GRAPH_SIZE - 1; i >= 1; i-- ) {
        s.push( i );
        //cout << i << " ";
    }
    //cout << s.size() << endl;
    while ( !s.empty() ) {
        city current = s.top();
        //cout << current << endl;
        s.pop();
        if ( current == NO_CITY ) {
            curr_tour.Remove_last_city();  //Remove last city( curr tour );
        }                                  // End of child list, back up
        else {
```

```
            curr_tour.Add_city( current );
            //cout << curr_tour.cost << endl;
            if ( curr_tour.route.size() == GRAPH_SIZE ) {
                int temp_cost = curr_tour.cost;
                curr_tour.Add_city( 0 );
                // for ( int i = 0; i < curr_tour.route.size(); i++ ) {
                //     cout << curr_tour.route[ i ] << " ";
                // }
                // cout << endl
                //      << curr_tour.cost << endl;
                //int cost = curr_tour.cost;
                if ( best.route.empty() || best.cost > curr_tour.cost ) {
                    best = curr_tour;
                }
                curr_tour.cost = temp_cost;
                curr_tour.route.pop_back();
                //cout << curr_tour.cost << endl;
                curr_tour.Remove_last_city();
                //cout << curr_tour.cost << endl;
            }
            else {
                s.push( NO_CITY );
                for ( city i = GRAPH_SIZE - 1; i >= 1; i-- ) {
                    if ( find( curr_tour.route.begin(), curr_tour.route.end(), i ) ==
curr_tour.route.end() && curr_tour.cost + graph[ curr_tour.route.back() ][ i ] <
best.cost ) {
                        s.push( i );
                    }
                }
            }
        }
        // if Feasible
    }
    return best;
    // while !Empty
}
```

第三种也是使用队列的迭代，其中栈中存储的接下来要搜索的环游(tour)，一次推入或弹出一个环游：

```
1  Push_copy(stack, tour);  // Tour that visits only the hometown
2  while (!Empty(stack)) {
3     curr_tour = Pop(stack);
4     if (City_count(curr_tour) == n) {
5        if (Best_tour(curr_tour))
6           Update_best_tour(curr_tour);
7     } else {
8        for (nbr = n-1; nbr >= 1; nbr--)
9           if (Feasible(curr_tour, nbr)) {
10             Add_city(curr_tour, nbr);
11             Push_copy(stack, curr_tour);
12             Remove_last_city(curr_tour);
13          }
14    }
15    Free_tour(curr_tour);
16 }
```

Program 6.6: Pseudocode for a second solution to TSP that doesn't use recursion

具体实现的函数代码如下：

```
Tour tsp() {
    stack<Tour> s;
    s.empty();
    Tour tour, best, curr_tour;
    tour.cost = 0;
    tour.route.clear();
    tour.route.push_back( 0 );
    best.cost = INT_MAX;
    best.route.clear();
    s.push( tour );
    //cout << s.size() << endl;
    while ( !s.empty() ) {
        curr_tour = s.top();
        s.pop();
        if ( curr_tour.route.size() == GRAPH_SIZE ){
            curr_tour.Add_city( 0 );
            for ( int i = 0; i < curr_tour.route.size(); i++ ) {
                cout << curr_tour.route[ i ] << " ";
            }
            cout << endl
                 << curr_tour.cost << endl;
            //int cost = curr_tour.cost;
            if ( best.cost > curr_tour.cost ) {
                best = curr_tour;
            }
```

```
        }
        else{
            for ( city i = GRAPH_SIZE - 1; i >= 1; i-- ) {
                if ( curr_tour.Feasible(best, i)) {
                    curr_tour.Add_city( i );
                    s.push( curr_tour );
                    curr_tour.Remove_last_city();
                }
            }
        }
        // if Feasible
    }
    return best;
    // while !Empty
}
```

2. Pthreds 并行化版本

对串行版本的并行化分为两种，一种是静态的并行，一种是动态的并行。静态的并行将搜索树静态地划分给每个线程去搜索，而动态的实现则采用动态负载均衡的方式，每个线程在运行完自己分配到的搜索任务后可以获取一些还没有被分配的任务来运行。本实验主要实现了静态并行的 Pthreads 版本，伪代码如下：

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

Program 6.7: Pseudocode for a Pthreads implementation of a statically parallelized solution to TSP

实现的部分函数代码如下：

```
void Partition_tree( long& my_rank, stack<Tour>& s ) {
    int my_first_tour, my_last_tour;
```

```
    if ( my_rank == 0 ) {
        Build_initial_queue();
    }
    Barrier();
    Set_init_tours( my_rank, &my_first_tour, &my_last_tour );
    for ( int i = my_last_tour; i >= my_first_tour; i-- ) {
        s.push( q[i] );
    }
}

void* Tsp( void* rank ) {
    long my_rank = (long)rank;
    stack<Tour> s;
    stack<Tour> available;
    Tour curr_tour;

    Partition_tree( my_rank, s );

    while ( !s.empty() ) {
        curr_tour = s.top();
        s.pop();
        if ( curr_tour.route.size() == GRAPH_SIZE ) {
            if ( curr_tour.cost + graph[curr_tour.route.back()][0] >
best.cost ){
                continue;
            }
            curr_tour.Add_city( 0 );
            if ( best.cost > curr_tour.cost ) {
                Update_best_tour( curr_tour );
            }
        }
        else {
            for ( city i = GRAPH_SIZE - 1; i >= 1; i-- ) {
                if ( curr_tour.Feasible( best, i ) ) {
                    curr_tour.Add_city( i );
                    s.push( curr_tour );
                    curr_tour.Remove_last_city();
                }
            }
        }
    }
    // if Feasible
    return NULL;
}
```

3. OpenMP 的并行化版本

使用 OpenMP 的并行化和使用 Pthreads 的并行化类似，实现的代码如下:

```
void Tsp() {
    int my_rank = omp_get_thread_num();
    stack<Tour> s;
    stack<Tour> available;
    Tour curr_tour;
    curr_tour.cost = 0;
    curr_tour.route.clear();
    best.cost = INT_MAX;

    Partition_tree( my_rank, s );

    while ( !s.empty() ) {
        curr_tour = s.top();
        s.pop();
        if ( curr_tour.route.size() == GRAPH_SIZE ) {
            if ( curr_tour.cost + graph[ curr_tour.route.back() ][ 0 ] >
best.cost ) {
                continue;
            }
            curr_tour.Add_city( 0 );
            // for ( int i = 0; i < curr_tour.route.size(); i++ ) {
            //     cout << curr_tour.route[ i ] << " ";
            // }
            // cout << endl
            //      << curr_tour.cost << endl;
            //int cost = curr_tour.cost;
            #pragma omp critical
            Update_best_tour( curr_tour );
        }
        else {
            for ( city i = GRAPH_SIZE - 1; i >= 1; i-- ) {
                if ( curr_tour.Feasible( best, i ) ) {
                    curr_tour.Add_city( i );
                    s.push( curr_tour );
                    curr_tour.Remove_last_city();
                }
            }
        }
    }
}
```

3. 实验结果与分析

编写好代码之后，就需要找到合适规模的 TSP 数据进行验证。由于设备资源和性能有限，一开始使用了太大的数据集导致运行时间缓慢，最后选用了如下的 TSP 数据来测试代码：

```
0   29  20  21  16  31  100 12  4   31  18
29  0   15  29  28  40  72  21  29  41  12
20  15  0   15  14  25  81  9   23  27  13
21  29  15  0   4   12  92  12  25  13  25
16  28  14  4   0   16  94  9   20  16  22
31  40  25  12  16  0   95  24  36  3   37
100 72  81  92  94  95  0   90  101 99  84
12  21  9   12  9   24  90  0   15  25  13
4   29  23  25  20  36  101 15  0   35  18
31  41  27  13  16  3   99  25  35  0   38
18  12  13  25  22  37  84  13  18  38  0
```

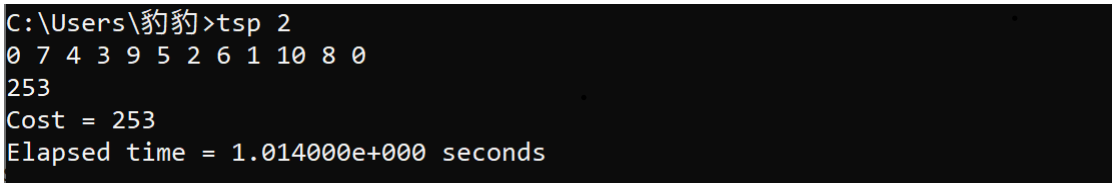
这个数据集的规模是 11 个城市，最优解的长度为 253，其中一个最优解为：

```
0   7   4   3   9   5   2   6   1   10  8   0
```

实验环境分别为个人电脑的 Windows 10 家庭版主机和 Ubuntu 21.10 虚拟机，其中主机的处理器为 Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz　1.50 GHz，共 8 个核心，虚拟机软件为 VMWare Workstation Player 16，分配了 2 个核。下表为在不同系统下的串行版本，OpenMP 版本和 Pthreads 版本测量到的运行时间：

Threads	Serial2 (Ubuntu)	Serial3 (Ubuntu)	OpenMP (Windows)	OpenMP (Ubuntu)	Pthreads (Ubuntu)
1	2.051910e+00 seconds	2.122171e+00 seconds	2.333000e+000 seconds	2.316053e+00 seconds	2.225714e+00 seconds
2			1.014000e+000 seconds	1.512324e+00 seconds	1.687260e+00 seconds
4			7.059999e-001 seconds	1.625435e+00 seconds	1.584839e+00 seconds
8			4.979999e-001 seconds	1.639739e+00 seconds	1.597318e+00 seconds

下面是部分在测试过程中的截图：



```
ocelot@ubuntu:~/final-project$ ./tsp_serial
0 7 4 3 9 5 2 6 1 10 8 0
253
Cost = 253
Elapsed time = 2.051910e+00 seconds
```

从表中可以看到，各个版本的并行程序在单线程运行时和串行版本的运行时间类似，而在线程数量增加时运行时间的减少情况有所不同。OpenMP 版本在 Windows 主机下加速效果最好，4 个线程和 8 个线程时加速比分别达到 3.3 和 2.0，效率分别为 0.825 和 0.5。而同样的 OpenMP 程序放到 Ubuntu 虚拟机下运行，就只有 2 个线程时运行速度有明显提升，Pthreads 版本情况也类似。之所以出现这种现象，推测可能是因为 VM Ware Workstation Player 虚拟机软件只能给 Ubuntu 虚拟机分配两个 CPU，所以超过 2 个线程时，运行速度就停止提升了。而且由于线程增加导致并行开销增大，程序需要花更多时间用于创建线程和分配任务，运行时间可能会更长（Ubuntu 下的 OpenMP）。

4. 实验总结

本次实验是并行与分布式计算的期末项目，在完成实验的过程中我系统地复习了 OpenMP 和 Pthreads 的编程并进行了实际练习，在实现书本上的例子的同时回顾了程序并行化的的设计思想与方法，体会到了前人的智慧与努力。同时我也锻炼了排查和解决问题，并且分析实验结果的能力，学会了如何与他人沟通，向他人请教等等。

在编写代码和运行的过程中，我还体会到了细节的重要性。尽管通常我们都是因为一些小的地方出现的问题而导致实验难以推进，但究其原因很多也是在大的方向上没有设计好，例如解决问题的整个思路和每一步具体需要完成什么目标等等。一旦大体的结构设计清晰，中途进行大改的可能性就会降低，细节处的问题很多也就迎刃而解了。

非常感谢老师和助教不厌其烦地回答我在实验中遇到的问题，希望自己在以后能够再接再厉，在并行与分布式计算和其他领域学到更多知识，解决更多问题。

5. 参考链接

1. https://stackoverflow.com/questions/10874214/measure-execution-time-in-c-openmp-code
2. https://stackoverflow.com/questions/11007355/data-for-simple-tsp
3. https://zh.wikipedia.org/zh-hans/%E6%97%85%E8%A1%8C%E6%8E%A8%E9%94%80%E5%91%98%E9%97%AE%E9%A2%98
4. https://zh.wikipedia.org/wiki/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2