## Description

> ***TSP问题（任选一个数据集）***
>
> *大规模输入文件为tsp.txt，每一行为每个城市的坐标*
> *小规模输入文件为tsp2.txt，文件内是城市之间的距离矩阵*
> *补充说明：可以自行调整数据规模，例如删减至10个城市，提交时需要将输入文件也一起打包*
> *输出文件为经过城市的顺序（从0开始编号）*

## Solution

### 0.问题再现

> *假设有一个旅行商要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。*

### 1. 算法选择

- TSP问题是个NP完全问题，其求解算法主要包括：
  1. DFS、BFS穷举搜索树，时间复杂度为$O(n!)$
  2. 状压DP，时间复杂度为$O(2^n n^2)$
  3. 一些随机的、启发式的搜索算法，比如遗传算法、蚁群算法、模拟退火算法、粒子群算法等，能把时间复杂度优化到一个多项式时间
- 课本上的方法即用DFS穷举搜索树，这种方法十分好写，而且课本已经给出了并行方案，所以一开始我也是采用此种方法的。

  然而，此种算法时间复杂度实在太高了，不剪枝的话，即使是跑老师给的小数据集，`17!`仍然是一个天文数字（$17! = 3.5e14 = 3e6$秒 $= 34$天），并行了也好不到哪里去。

  然后就尝试了各种剪枝，如剪去已经大于当前最小值的枝等，优化完也仍然较慢，用`OpenMP`四线程跑小数据集十几分钟跑了百分之三十。这样子并行优化还不如算法优化，因此放弃了暴搜+剪枝的方案
- 然后考虑模拟退火等启发式算法，首先这些算法对我来说实现起来还是比较麻烦的，而且也暂时想不到什么好方法并行。关键是这些启发式算法还有个最大缺点，就是只能求近似最优解，不能保证一定能求出最优解，所以也放弃了这种方法。
- 因此，最后采用了状压DP的方法，时间复杂度为$O(2^n n^2)$，跑个规模17的数据集还是很快的$2^{17} * 17^2 = 37879808$，串行理论时间复杂度不到一秒，虽然规模48那个数据集跑起来还是要几天几夜（那个数据集要跑感觉只能用遗传算法求近似最优解了）。

### 2. 串行算法思路及实现

- 状压DP，即状态压缩动态规划，是利用计算机二进制的性质来描述状态的一种DP方式。 要实现算法，一是要找好状态转移方程，二是要用二进制描述好每个状态。
- 推导状态转移方程：

  假设从顶点s出发，令dp(i, State)表示从顶点i出发经过State(是一个点的集合)中各个顶点一次且仅一次，最后回到出发点s的最短路径长度。现在开始推导其状态转移方程：

1. 当V为空集，那么$dp(i, State)$，表示直接从i回到s了，此时$dp(i, State) = c_{is}(i \neq s)$
2. 如果State不为空，那么就是对子问题的最优求解。必须在State这个城市集合中，尝试每一个，并求出最优解。$dp(i, State) = min(C_{ik} + dp(k, State - (k)))$

以上即为算法的状态转移方程。另外，不难发现最终路径是个环，所以不论出发点定在哪里，结果都是一样的。所以为了简化过程，不妨假设起点都为0号城市。

- 利用二进制描述状态：

  考虑dp数组的第二维State，其含义是经过城市编号的点集，则自然想到若有N个城市，能用一个N-1位二进制数来表示每种状态（无需考虑出发点），该二进制数的第n位是否为1表示第n个城市是否在集合中。例如，对于N=4，集合{1,3}可表示为二进制数101，对应十进制数即为3，因此，第二维可用一个$2^{n-1}$规模的十进制数表示。根据二进制位运算，可以定义以下操作：

  - 对于第y个城市，他的二进制表达为，1<<(y-1)；
  - 对于数字x，要看它的第i位是不是1，那么可以通过判断布尔表达式 (((x >> (i - 1) ) & 1) == 1的真值来实现；
  - 由动态规划公式可知，需要从集合中剔除元素。假如集合用索引x表示，要剔除元素标号为i,我们异或运算实现减法，其运算表示为： x = x ^ (1<<(i - 1))。
- 数据结构、全局变量定义：

```
#define N 17            // 城市数量
#define INF 1 << 30  // 定义最大值，  用于tsp更新dp数组

// 用于计算时刻
#define GET_TIME(now)                              \
    {                                              \
        struct timeval t;                          \
        gettimeofday(&t, NULL);                    \
        now = t.tv_sec + t.tv_usec / 1000000.0; \
    }

const int M = 1 << (N - 1);  // 状态数 M = 2^(n-1)
int g[N][N];                 // 存原始图
int dp[N][M];  // dp数组，dp[point][state]表示从point出发经过state中各个点一次且仅一次，最后回到出发点的最短路径长度
vector<int> path;  // 路径数组，用于打印路径
```

- 求解TSP函数主体：

```
/**
 * @brief 串行版本的tsp求解函数，用状压dp实现，理论复杂度为O((2^n)*(n^2))，由于路径是一个环，设出发点为0号城市
 * @param void
 * @return void
 * @author 黄海宇
 * @date 2021/5/3
 */
void Serial_TSP() {
    // 初始化，不经过任何点的dp数组状态置为g[i][0]，即该点到出发点0号城市之间的距离
    for (int i = 0; i < N; i++) {
        dp[i][0] = g[i][0];
    }
    // 进入状态处理循环，第一层for，遍历状态维，即经过城市集合
    for (int j = 1; j < M; j++) {
```

```
        // 第二层for，遍历当前城市
        for (int i = 0; i < N; i++) {
            // 初始化当前状态最小路径长度为inf
            dp[i][j] = INF;
            // 如果要经过路径上的城市包括当前城市，违规，进入下一次循环
            if (((j >> (i - 1)) & 1) == 1) {
                continue;
            }
            // 遍历转移自哪个状态，取min{c_ik+d(k,V-{k})};
            for (int k = 1; k < N; k++) {
                if (((j >> (k - 1)) & 1) == 0) {
                    continue;
                }
                if (dp[i][j] > g[i][k] + dp[k][j ^ (1 << (k - 1))]) {
                    // 状态转移
                    dp[i][j] = g[i][k] + dp[k][j ^ (1 << (k - 1))];
                }
            }
        }
    }
}
```

- 根据dp结果找到路径并将其打印

  可以用dp数组来反向推出其路径。其算法思想如下:

  比如在第一步时，我们就知道那个值最小，因为dp[][]数组我们已经计算出来了，由计算可知$C_{01}+d(1,2,3,4)$最小，所以一开始从起始点0出发，经过1。接下来同样计算d(1,{2,3,4})，由计算可知$C_{14}+d(4,2,3)$所以0--->1---->4，接下来同理求d(4,{2,3})，以此类推，最终反向推出整条路径。

```
/**
 * @brief 用于判断是否所有城市都已经被访问过了
 * @param visited   传入访问记录数组
 * @return 所有城市都已经被访问过了，则返回1，否则返回0
 * @author 黄海宇
 * @date 2021/5/3
 */
bool isVisited(bool visited[]) {
    for (int i = 1; i < N; i++) {
        if (visited[i] == 0) {
            return 0;
        }
    }
    return 1;
}

/**
 * @brief 用于根据dp结果找到最短路径并将其输出
 * @param void
 * @return void
 * @author 黄海宇
 * @date 2021/5/3
 */
```

```
void FindAndPrintPath() {
    bool visited[N] = {false};
    int par = 0, min = INF, S = M - 1, temp;
    path.push_back(0);

    while (!isVisited(visited)) {
        for (int i = 1; i < N; i++) {
            if (visited[i] == false && (S & (1 << (i - 1))) != 0) {
                if (min > g[i][par] + dp[i][(S ^ (1 << (i - 1)))]) {
                    min = g[i][par] + dp[i][(S ^ (1 << (i - 1)))];
                    temp = i;
                }
            }
        }
        par = temp;
        path.push_back(par);
        visited[par] = true;
        S = S ^ (1 << (par - 1));
        min = INF;
    }
    cout << "The path is: ";
    for (auto val:path) {
        cout << val << " -> ";
    }
    cout << 0;
}
```

- init函数用于接收输入邻接矩阵并转存到二维数组g中

```
/**
 * @brief 初始化函数，用于接受输入的邻接矩阵并将其存入g中
 * @param void
 * @return void
 * @author 黄海宇
 * @date 2021/5/3
 */
void init() {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            cin >> g[i][j];
        }
    }
}
```

### 3. 使用OpenMP的并行算法思路及实现

- 要实现并行化，首先要找到可并行化的部分，该部分要求一是不能有数据依赖，二是并行工作量不能太少，否则切换进程的时间还不如串行，可以看到串行版本的TSP函数主体求解时嵌套了三个for，那最好当然是并行最外层循环，然而，最外层循环用来遍历dp数组的状态维，存在数据依赖，不能直接拆分并行。因此，考虑对这层for进行分类，去除数据依赖。考虑State的意义，它是每个状态经过中间点的集合，可对集合中点的个数进行分类，可以证明，集合中元素相等的一类状态不会互相影响，一个状态只依赖于点集中点的数量减一的状态集。因此，可先以此对外层循环分类，分类后的每一类状态便可以用来并行了。

  因此下面关注两个点，一是对状态进行分类，二是利用OpenMP具体实现并行

- 对状态进行分类：

  点集中的点的个数，其实就是二进制数中1的个数，因此写一个小接口，用来计算N-1位二进制数中有n个1的所有数即可。将此接口设计为内联函数，以加快运行速度。

  用到的全局变量：

```
int b[M];  // 存放相同城市数的state集合的十进制表示（仅在两个多线程版本用到）
```

  接口代码：

```
/**
 * @brief 找到所有N位二进制数里面有n个1对应的十进制数，并存到b数组中，设置为inline函数，加快运行时速度
 * @param n     所求二进制数中1的个数
 * @return 返回b数组有效长度
 * @author 黄海宇
 * @date 2021/5/3
 */
inline int gather(int n) {
    bool first = 1; // 特判第一次进入循环
    int r, c;
    int cnt = 0;
    for (int j = (1U << n) - 1; first || r < (1 << (N - 1));) {
        b[cnt] = j; // 将得到的每个数存入b数组
        cnt++;
        c = j & -j;
        r = j + c;
        j = (((r ^ j) >> 2) / c) | r;
        first = 0;
    }
    return cnt;
}
```

- 利用OpenMP实现并行：

  OpenMP版本的并行实现十分简单，引入个头文件，在要并行的循环前加个`parallel for`即可，我的for里也没有要特殊处理的私有变量，因此处理起来十分简单。

  OpenMP并行版本函数主体：

```
/**
 * @brief 使用openmp的并行版本的tsp求解函数，并行部分为state集合中定点数相同的dp数组状态的处理
```

```
 * @param void
 * @return void
 * @author 黄海宇
 * @date 2021/5/3
 */
void openmp_parallel_TSP() {
    for (int i = 0; i < N; i++) {
        dp[i][0] = g[i][0];
    }
    // 对经过的城市集合进行分类，经过城市数量相同的归为同一类
    for (int n = 1; n < N; n++) {
        // 求出同类情况，并放入b数组中，b数组有效长度存入cnt
        int cnt = gather(n);
        // 同一类之间没有数据依赖，因此可以并行处理同一类类情况的状态
        #pragma omp parallel for num_threads(thread_count)
        for (int index = 0; index < cnt; ++index) {
            int j = b[index];
            for (int i = 0; i < N; i++) {
                dp[i][j] = INF;
                if (((j >> (i - 1)) & 1) == 1) {
                    continue;
                }
                for (int k = 1; k < N; k++) {
                    if (((j >> (k - 1)) & 1) == 0) {
                        continue;
                    }
                    if (dp[i][j] > g[i][k] + dp[k][j ^ (1 << (k - 1))]) {
                        dp[i][j] = g[i][k] + dp[k][j ^ (1 << (k - 1))];
                    }
                }
            }
        }
    }
}
```

- 1
- 1

## 4. 使用Pthread的并行算法思路及实现

- Pthread版本的并行思路与OpenMP版本相同，仍是考虑对外层for进行分类，将点集中元素个数相同的每一类状态进行并行处理。不过Pthread版本并行实现起来较为复杂，需要设计线程函数，并自己设计负载均衡策略。
- 用到的全局变量和数据结构：

```
int b[M];  // 存放相同城市数的state集合的十进制表示（仅在两个多线程版本用到）

/**
 * @brief 该结构体用于存放向pthread的线程函数传入的多个参数
 * @author 黄海宇
 * @date 2021/5/3
 */
struct parameter {
    void* rank;  // 线程号
    int* ar;     // 待划分的数组
    int len;     // 待划分的数组的长度
};
```

由于b数组一直在发生变化，要创建多次线程，因此线程函数的参数要传入不止一个，而`pthread_create()`函数用于传递线程函数参数的参数只有一个，因此考虑而外设计`parameter`结构体，用于将待传入参数进行打包。实际要传入的参数包括线程号、待划分的数组以及待划分的数组长度。

- 利用`Pthread`实现并行的函数主体:

```
/**
 * @brief 使用pthread的并行版本的tsp求解函数，并行部分为state集合中定点数相同的dp数组状态的处理
 * @param void
 * @return void
 * @author 黄海宇
 * @date 2021/5/3
 */
void pthread_parallel_TSP() {
    for (int i = 0; i < N; i++) {
        dp[i][0] = g[i][0];
    }
    // 对经过的城市集合进行分类，经过城市数量相同的归为同一类
    for (int n = 1; n < N; n++) {

        // 求出同类情况，并放入b数组中，b数组有效长度存入cnt
        int cnt = gather(n);
        // 同一类之间没有数据依赖，因此可以并行处理同一类类情况的状态

        long thread; // 线程号
        pthread_t* thread_handles; // 线程指针

        // 为线程指针申请内存
        thread_handles = (pthread_t*)malloc(thread_count *
sizeof(pthread_t));

        // 开启thread_count个线程，每个线程启用线程函数
        for (thread = 0; thread < thread_count; thread++) {
            struct parameter* par = new parameter;
            par->rank = (void*)thread;
            par->len = cnt;
            par->ar = b;
            pthread_create(&thread_handles[thread], NULL, Thread_work,
                           (void*)par);
```

```
        }

        // 同步线程
        for (thread = 0; thread < thread_count; thread++)
            pthread_join(thread_handles[thread], NULL);

        free(thread_handles);
    }
}
```

逻辑上，并行策略与OpenMP相同。实际上，函数主体主要包括以下工作：

1. 每次对状态进行分类；
2. 每次分类后开启thread_count个线程，所以一共需要开启N*thread_count次线程；
3. 对于开启的每个线程，先将待传入参数打包到一个parameter结构体中，再通过pthread_create调用线程函数；
4. 同步线程，释放之前申请的空间

- 线程函数工作：

```
/**
 * @brief pthread的线程函数
 * @param arg 打包多个参数的参数结构体，包括线程号、待划分数组、待划分数组的长度
 * @return void*
 * @author 黄海宇
 * @date 2021/5/3
 */
void* Thread_work(void* arg) {
    // 解引用传入的参数结构体
    struct parameter* p = (parameter*)arg;
    void* rank = p->rank;
    int* ar = p->ar;
    int n = p->len;
    free(p);
    long my_rank = (long)rank;

    // 计算各线程负责处理的数组区间的左右值，尽量负载均衡
    long long my_n = n / thread_count;
    long long res = n % thread_count;
    long long my_left;
    long long j;

    if (my_rank < res) {
        my_n++;
        my_left = my_n * my_rank;
    } else {
        my_left = my_n * my_rank + res;
    }
    long long my_right = my_left + my_n;

    // 各线程处理各自负责部分的数组
    for (int index = my_left; index < my_right; ++index) {
        j = ar[index];
        for (int i = 0; i < N; i++) {
            dp[i][j] = INF;
            if (((j >> (i - 1)) & 1) == 1) {
```

```
                continue;
            }
            for (int k = 1; k < N; k++) {
                if (((j >> (k - 1)) & 1) == 0) {
                    continue;
                }
                if (dp[i][j] > g[i][k] + dp[k][j ^ (1 << (k - 1))]) {
                    dp[i][j] = g[i][k] + dp[k][j ^ (1 << (k - 1))];
                }
            }
        }
    }
    return nullptr;
}
```

线程函数的主要工作有：

1. 解引用传入的`parameter`结构体，将其空间进行回收；
2. 计算本线程所负责区间的左右端点。为实现负载均衡，首先将总的待处理数组长度均分，再将所剩余数`res`分配到前`res`个线程中；
3. 各线程处理各自负责部分的数组。由于各线程更新dp数组的不同位置处，因此不存在共享变量导致的临界区问题，因此也不需加锁。
4. 最后返回`nullptr`，一开始没注意要有返回值导致开`-O2`编译参数时出现一些奇怪的问题。

## 5. 程序执行及运行结果显示

- 为便于比较，将三个版本的解法均写入同一个程序中。主函数实现如下：

```
int main() {
    // 计时开始时刻和计时结束时刻
    double start, stop;
    init();

    // 串行版本:
    cout << "1. Serial Version:" << endl;
    GET_TIME(start);
    Serial_TSP();
    GET_TIME(stop);
    cout << "The minimal distance: " << dp[0][M - 1] << endl;
    printf("Serial run time: %e\n\n", stop - start);
    FindAndPrintPath();
    cout << endl << endl;

    cout << "---------------------------------" << endl << endl;

    // 重置dp数组和path，不影响下个版本的工作
    memset(dp, 0, sizeof(dp));
    path.clear();

    // OpenMP版本:
    cout << "2. OpenMP Version:" << endl;
    GET_TIME(start);
    openmp_parallel_TSP();
    GET_TIME(stop);
```

```
    cout << "The minimal distance: " << dp[0][M - 1] << endl;
    printf("parallel with openmp run time: %e\n\n", stop - start);
    FindAndPrintPath();

    cout << endl << endl;

    cout << "--------------------------------" << endl << endl;

    // 重置dp数组和path，不影响下个版本的工作
    memset(dp, 0, sizeof(dp));
    path.clear();

    // Pthread版本:
    cout << "3. Pthread Version:" << endl;
    GET_TIME(start);
    pthread_parallel_TSP();
    GET_TIME(stop);
    cout << "The minimal distance: " << dp[0][M - 1] << endl;
    printf("parallel with pthread run time: %e\n\n", stop - start);
    FindAndPrintPath();

    return 0;
}
```

值得注意的是，为了避免上个版本的运行结果影响下个版本，需在两个版本的调用间重置dp数组和path。

- 编译：

  由于本程序同时包括OpenMP版本和Pthread版本，因此两个动态链接库都要加入编译参数中

```
g++ -Wall -g -o  tsp tsp.cpp -fopenmp -lpthread
```

- 运行：

  改变输入输出流，重定向到文件：

```
./tsp  <输入文件名  >输出文件名
```

- 运行结果举例：

  数据集：tsp2.txt，17个城市

  线程数：4

```
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL  ~/workspace/hpc/mid_project  ./tsp <tsp2.txt
                                  1150  10:36:03
1. Serial Version:
The minimal distance: 2085
Serial run time: 8.793497e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8
-> 11 -> 15 -> 0

--------------------------------

2. OpenMP Version:
The minimal distance: 2085
parallel with openmp run time: 1.916814e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8
-> 11 -> 15 -> 0

--------------------------------

3. Pthread Version:
The minimal distance: 2085
parallel with pthread run time: 2.897620e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8
-> 11 -> 15 -> 0
```

经验证，计算结果正确。

串行时间`0.087s`，四线程`OpenMP`时间`0.019s`，四线程`Pthread`时间`0.028s`，并行优化效果明显

## 6. 性能比较与分析

- 测试环境：联想小新Air13 Laptop 8处理器
- 测试方法：改变数据集规模和线程数，纵向和横向比较串行方案、`OpenMP`并行方案、`Pthread`并行方案的性能
- 部分测试过程：

  `thread_count = 4, N = 5:`

```
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL  ~/workspace/hpc/mid_project  g++ -Wall -g -o  tsp tsp.cpp -
fopenmp -lpthread     1181  11:40:15
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL  ~/workspace/hpc/mid_project  ./tsp <tsp3.txt
                                  1182  11:40:18
1. Serial Version:
The minimal distance: 23
Serial run time: 5.006790e-06

The path is: 0 -> 1 -> 4 -> 2 -> 3 -> 0

--------------------------------

2. OpenMP Version:
The minimal distance: 23
parallel with openmp run time: 1.509190e-04

The path is: 0 -> 1 -> 4 -> 2 -> 3 -> 0

--------------------------------

3. Pthread Version:
The minimal distance: 23
parallel with pthread run time: 2.091885e-03

The path is: 0 -> 1 -> 4 -> 2 -> 3 -> 0
```

数据规模较小时，并行处理带来的增益小于线程切换带来的开销，因此串行版本反而更快。

`thread_count = 8, N = 17:`

```
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL ~/workspace/hpc/mid_project g++ -Wall -g -o tsp tsp.cpp -
fopenmp -lpthread
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL ~/workspace/hpc/mid_project ./tsp <tsp2.txt
1. Serial Version:
The minimal distance: 2085
Serial run time: 9.121799e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 ->
 0

--------------------------------

2. OpenMP Version:
The minimal distance: 2085
parallel with openmp run time: 3.342199e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 ->
 0

--------------------------------

3. Pthread Version:
The minimal distance: 2085
parallel with pthread run time: 3.017402e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 ->
 0%
```

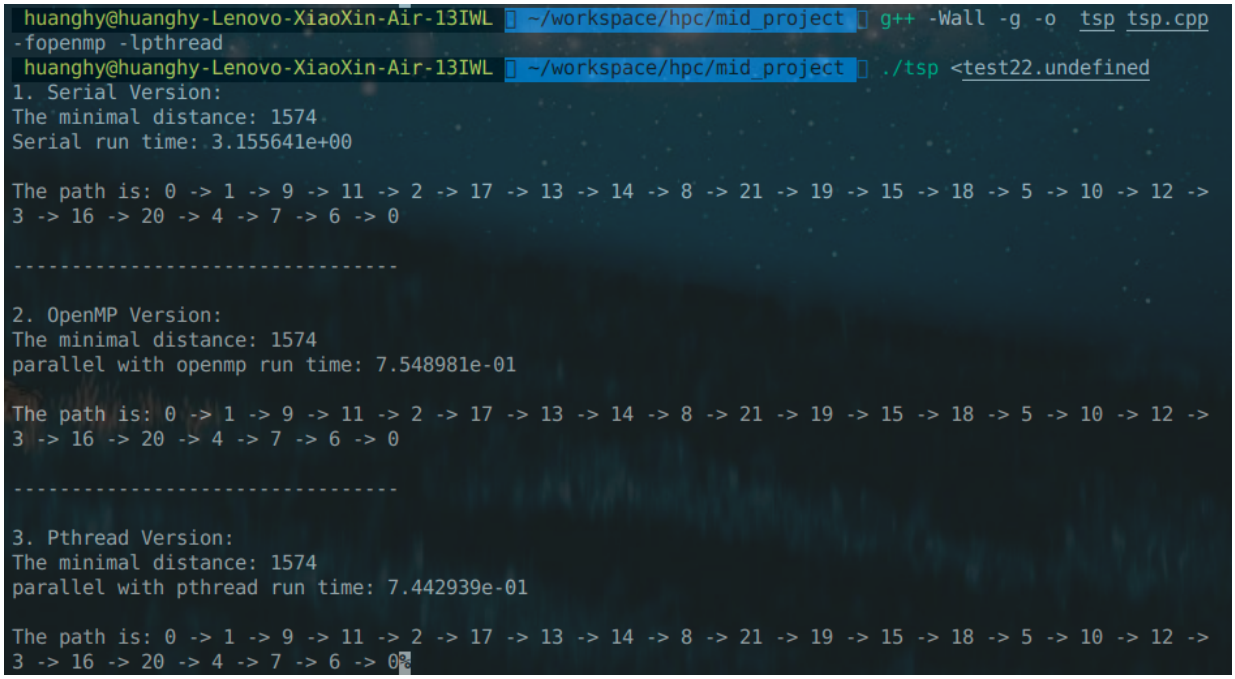
数据规模较大时，并行优化效果开始明显，OpenMP与Pthread运行性能不相上下，并行运行时间约为串行的32%

`thread_count = 4, N = 22:`

```
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL ~/workspace/hpc/mid_project g++ -Wall -g -o tsp tsp.cpp
-fopenmp -lpthread
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL ~/workspace/hpc/mid_project ./tsp <test22.undefined
1. Serial Version:
The minimal distance: 1574
Serial run time: 3.155641e+00

The path is: 0 -> 1 -> 9 -> 11 -> 2 -> 17 -> 13 -> 14 -> 8 -> 21 -> 19 -> 15 -> 18 -> 5 -> 10 -> 12 ->
3 -> 16 -> 20 -> 4 -> 7 -> 6 -> 0

--------------------------------

2. OpenMP Version:
The minimal distance: 1574
parallel with openmp run time: 7.548981e-01

The path is: 0 -> 1 -> 9 -> 11 -> 2 -> 17 -> 13 -> 14 -> 8 -> 21 -> 19 -> 15 -> 18 -> 5 -> 10 -> 12 ->
3 -> 16 -> 20 -> 4 -> 7 -> 6 -> 0

--------------------------------

3. Pthread Version:
The minimal distance: 1574
parallel with pthread run time: 7.442939e-01

The path is: 0 -> 1 -> 9 -> 11 -> 2 -> 17 -> 13 -> 14 -> 8 -> 21 -> 19 -> 15 -> 18 -> 5 -> 10 -> 12 ->
3 -> 16 -> 20 -> 4 -> 7 -> 6 -> 0%
```

数据规模越大，并行优化效果越明显，OpenMP与Pthread运行性能仍然不相上下，并行运行时间约为串行的26%

`thread_count = 8, N = 22:`




数据规模保持不变时，线程数增加，并行优化效果越明显，OpenMP与Pthread运行性能仍然不相上下，并行运行时间约为串行的22%

- 结论：
    1. 数据规模在30以下时，三种版本的性能表现均可接受
    2. 数据规模较小时，并行优化效果不明显
    3. 随着数据规模越来越大，并行优化效果越来越明显，当数据规模达到25时，八线程并行运行时间为串行的约16%
    4. 数据规模保持不变时，随着线程数增加，并行优化效果越来越明显

## 7.加入其他优化策略

- 为了使性能再上一层楼，在原来并行策略上又加入了编译指令的O2优化，以及对内层for进行循环展开
- O2优化：

  原理是编译器提供的满足用户优化需要的选项，包括：

  1）精简操作指令；2）尽量满足cpu的流水操作；3）通过对程序行为地猜测，重新调整代码的执行顺序；4）充分使用寄存器

  此优化策略实现起来十分简单，稍微修改编译指令，加入-O2编译参数即可：

```
g++ -O2 -Wall -g -o  tsp tsp.cpp -fopenmp -lpthread
```

  优化效果：

  未开启O2：

```
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL ~/workspace/hpc/mid_project g++ -Wall -g -o tsp tsp.cpp -
fopenmp -lpthread
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL ~/workspace/hpc/mid_project ./tsp <tsp2.txt
1. Serial Version:
The minimal distance: 2085
Serial run time: 9.121799e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 ->
 0

--------------------------------

2. OpenMP Version:
The minimal distance: 2085
parallel with openmp run time: 3.342199e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 ->
 0

--------------------------------

3. Pthread Version:
The minimal distance: 2085
parallel with pthread run time: 3.017402e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 ->
 0
```

开启O2:

```
huanghy@huanghy-Lenovo-XiaoXin-Air-13IWL ~/workspace/hpc/mid_project ./tsp <tsp2.txt
1. Serial Version:
The minimal distance: 2085
Serial run time: 4.838610e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 -
> 0

--------------------------------

2. OpenMP Version:
The minimal distance: 2085
parallel with openmp run time: 9.988070e-03

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 -
> 0

--------------------------------

3. Pthread Version:
The minimal distance: 2085
parallel with pthread run time: 2.138495e-02

The path is: 0 -> 3 -> 12 -> 6 -> 7 -> 5 -> 16 -> 13 -> 14 -> 2 -> 10 -> 9 -> 1 -> 4 -> 8 -> 11 -> 15 -
> 0
```

这种方法串行并行一起优化了，能将运行时间减少到原来的50%左右

- 循环展开:

  循环展开即手动展开嵌套在内部的循环，能够增大指令调度的空间，减少循环分支指令的开销，可以更好地实现数据预取技术。

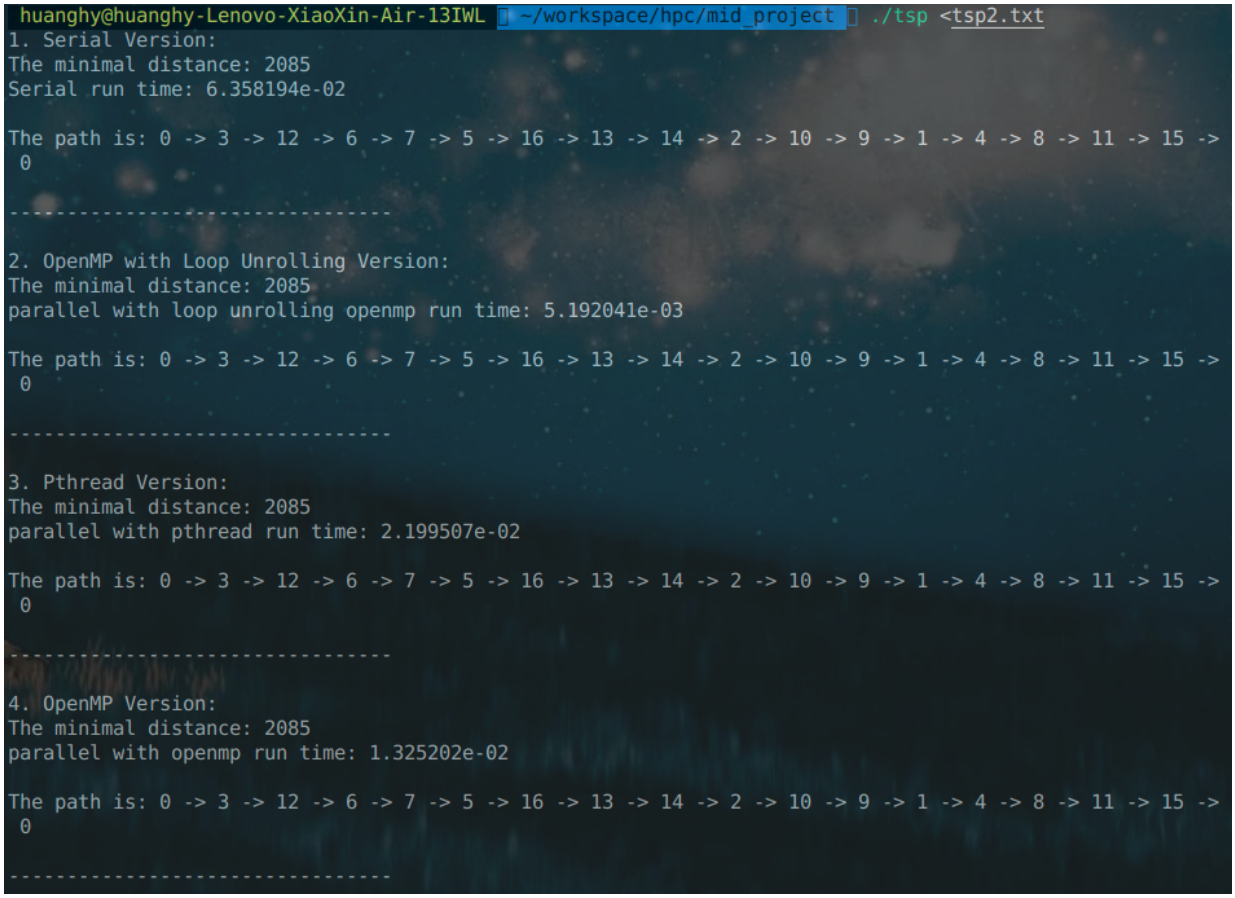
  下面对OpenMP的并行版本的最内层循环进行16次循环展开:

```
/**
 * @brief 设置为内联函数，用于循环展开
 * @param void
 * @return void
 * @author 黄海宇
 * @date 2021/5/3
 */
inline void unroll_loop(int i, int j, int n) {
    if (((j >> (n - 1)) & 1) != 0) {
        if (dp[i][j] > g[i][n] + dp[n][j ^ (1 << (n - 1))]) {
            dp[i][j] = g[i][n] + dp[n][j ^ (1 << (n - 1))];
```

```
        }
    }
}

/**
 * @brief 使用循环展开的openmp的并行版本
 * @param void
 * @return void
 * @author 黄海宇
 * @date 2021/5/3
 */
void openmp_unroll_loop_parallel_TSP() {
    for (int i = 0; i < N; i++) {
        dp[i][0] = g[i][0];
    }
    // 对经过的城市集合进行分类，经过城市数量相同的归为同一类
    for (int n = 1; n < N; n++) {
        // 求出同类情况，并放入b数组中，b数组有效长度存入cnt
        int cnt = gather(n);
// 同一类之间没有数据依赖，因此可以并行处理同一类类情况的状态
#pragma omp parallel for num_threads(thread_count)
        for (int index = 0; index < cnt; ++index) {
            int j = b[index];
            for (int i = 0; i < N; i++) {
                dp[i][j] = INF;
                if (((j >> (i - 1)) & 1) == 1) {
                    continue;
                }
                unroll_loop(i,j,1);
                unroll_loop(i,j,2);
                unroll_loop(i,j,3);
                unroll_loop(i,j,4);
                unroll_loop(i,j,5);
                unroll_loop(i,j,6);
                unroll_loop(i,j,7);
                unroll_loop(i,j,8);
                unroll_loop(i,j,9);
                unroll_loop(i,j,10);
                unroll_loop(i,j,11);
                unroll_loop(i,j,12);
                unroll_loop(i,j,13);
                unroll_loop(i,j,14);
                unroll_loop(i,j,15);
                unroll_loop(i,j,16);
            }
        }
    }
}
```

注意循环展开函数必须写成内联函数

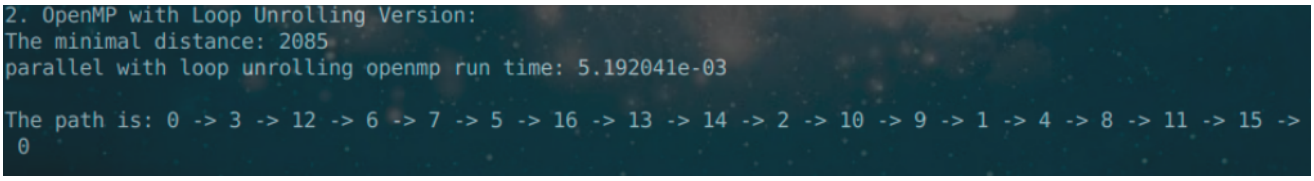
优化效果:




八线程数据规模为17的数据集用带循环展开的OpenMP能跑到0.005s，突破了一个数量级，运行时间又下降了约50%

## 8. 最好效果

用OpenMP并行化，线程数thread_count=8，编译参数开启-O2，循环展开16次，跑tsp2.txt数据集，运行最快时间为：




0.005192秒

## 9. 实验过程中遇到的困难和感想

1. 困难：

> *使用pthread_create时，之前线程函数一直只有一个，这次线程函数要传入的参数不止一个。*

解决方法：

> *自定义一个参数结构体，将要传的参数打包，这样pthread_create就只需要传一个参数了，到线程函数里再将结构体解引用。要注意传入的是指针，因此为了防止几个线程同时对其进行修改，因此没创建一个线程都要新创建一个参数结构体，在线程函数里再将其释放掉即可。*

2. 困难：

> *遇到很奇怪的情况：开-O2之后Pthread版本会发生段错误，但不开-O2却没事*

解决方法：

> *多线成debug十分麻烦，在经历了一系列波折后，发现竟然是线程函数没有返回值导致的，加了个`return nullptr`就好了，可见以后要吸取教训，不能靠默认返回值。*

- 感想：

这次实验着实让我收获颇丰，不仅让我更加熟练掌握了Pthread和OpenMP并行程序的编写方法，更让我培养了考虑优化算法性能、优化程序性能的能力。同时，通过踩一些坑，也能让我以后碰到这些情况能处理得更加得心应手。