高级数据结构-LAB2-PART2 实验分析报告

一．并行化思路：

1.选择的要进行并行化的算法部分：

通过分析，我决定对“**遍历所有全排列路径找到最短路径**”部分进行并行化处理。

2. 如此选择的理由：

通过 lab2-part1 的实践以及测试发现，我的算法主要时间消耗在两个部分，一是 Dijkstra 算法找到最短路径，一个就是遍历所有排列点集来找到最小值的过程。而对于这两个部分，通过我在 part1 部分的时间测试来看，第二部分占比更大，并且随着 n 的增大而指数级增长。这与理论上的遍历全排列的暴力算法的时间复杂度为 O(N*V*V)是相符的。所以我决定通过对第二部分进行并行化的方式来尝试改善程序的性能。

3. 如何具体实现并行化：

在初始算法的基础上，按照给定的 thread 的数量，将要遍历的所有情况分成 N 块。而考虑到 N 可能不是整数(即存在不能整除这种情况)，所以对于前 N-1 块区域，利用 for 循环进行遍历，并在每一次迭代创建一个 thread 来运行遍历这块区域的范围内的所有的路径全排列，并在最后在此创建一个 thread 来处理剩余的可能存在的没有整除的区域，在所有 thread 创建完成并执行结束之后利用 thread 的 join()函数进行 thread 的回收。之后记录上述过程的总时间作为并行之后的时间。

每个 thread 调用的主函数如下：

```
void getMinPath_parallel(vector<vector<int>> AllPaths, int &min_len, vector<int> &min_path
                         , int start, int end, int source);
```

上述 AllPaths 为记录所有全排列的 vector，min_len 与 min_path 分别记录最短路径值以及具体的最短路径，start 与 end 分别表示这个 thread 代表的部分的起始与结束的索引位置，source 是传递进来的起点数据。

而考虑到整个过程中虽然用到共享全局变量但是只涉及到访问而没有修改，另外也没有使用动态指针，所以就没有使用 lock 进行加锁。具体并行部分如下：

```
vector<thread> threads;
int N = 16;//记录线程数量
int block_size = (Num / N);//计算每一个thread应该处理的数据量
int start = 0;
for(int i = 0;i < (N);++i){
    int end = start + block_size - 1;
    threads.emplace_back(&FixedSP::getMinPath_parallel, this, AllPaths, ref(min_len), ref(min_path)
             , start, end, source);//创建线程(注意线程在进行参数引用的格式)
    start = end + 1;
}
for(size_t i = 0;i < threads.size();++i) threads[i].join();
```

之后在每个 thread 的主函数 getMinPath_Parallel 中测定每个 thread 的并行运行的时间，并取平均值作为最终的实验数据并记录。

二．性能测试以及结论：

1. 硬件环境：

联想 ThinkPad 笔记本电脑，core i7，四内核，最多可以同时运行 8 个线程

2. 测评方案设计：

I.线程数选择：分别选择 **1(单线程),2,4,8,16** 进行测试

II.数据集选择：由于主要是对全排列部分进行并行化处理，所以测试集的变化主要是来源于选择的中间点的个数 n。在本次实验中，我选择了 **n 分别为 5,6,7,8 四组**用于测试。

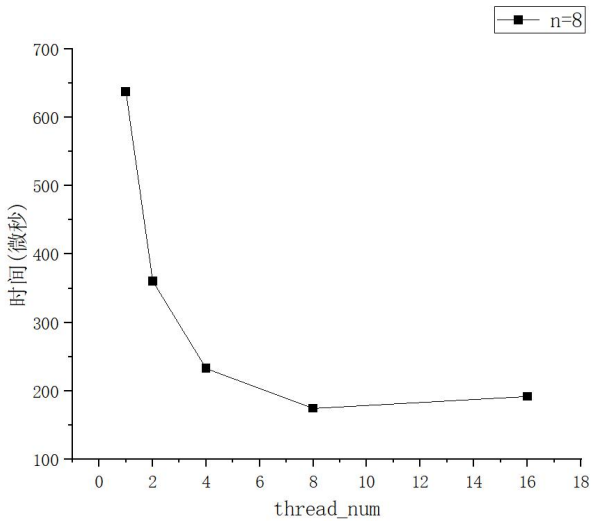
III.数据原始矩阵选择：选择的是 part1 中老师提供的测试 4 中的 **N=200 的矩阵**最为原始基本矩阵数据。

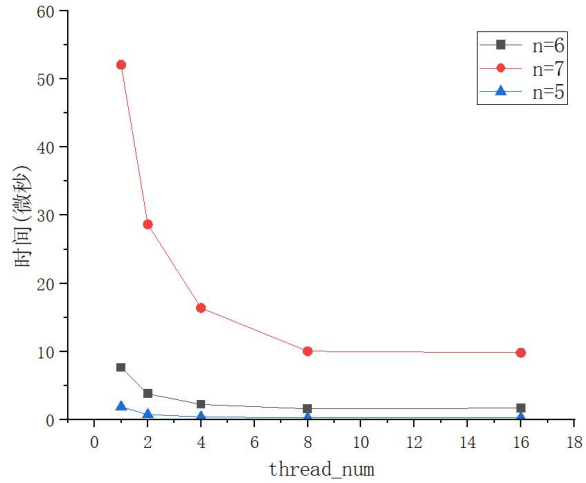
3.**测评结果：**

I.结果表：

运行时间	thread=1	thread=2	thread=4	thread=8	thread=16
n=5	1.864	0.737	0.387	0.238	0.276
n=6	7.627	3.771	2.219	1.574	1.681
n=7	52.093	28.662	16.389	10.035	9.819
n=8	637.911	360.532	232.621	174.113	191.614
注：表格中的时间以微秒为单位，并且为1e6数量级的数据					

II.结果图：(由于 n=8 组与 n=5,6,7 组的时间相差过大，故分开绘制曲线图以体现效果)





4.**实验结论：**

从上面的数据图表可以得出如下结论：**在线程数为 1/2/4/8 时，程序执行效率随着线程数量的增加而逐渐下降，而线程数为 16 个的时候，会出现性能持平甚至效率下降的情况。**

5.**结论分析：**

①在线程数为 1/2/4/8 时，虽然随着线程数量的增加性能逐渐上升，但是并不满足理论上的完全线性。可能原因如下：

I. 创建线程、输出运行时间等操作会增加程序运行时的 overhead，从而使得性能并不完全

符合理论上的线性变化趋势。

II. 实验时采用了每组数据测试 5 次取平均的方式，测试次数的不足可能是性能曲线不完全符合线性分布的原因。

②在线程数为 16 时，性能持平甚至出现下降的情况，可能的原因如下：

由于程序运行在最多同时运行 8 个 thread 的机器上面，所以当线程数量超过 8 时，同一个 CPU 上面要处理的线程至少为 2 个，则同核上面的 thread 之间竞争加剧，产生了额外的较大的 overhead，从而可能使得程序执行性能出现上述的持平甚至下降的情况。