过定点最短路中Dijkstra算法的并行化实验报告

并行化思路

在我的串行执行版本中，在一次节点个数为V,必经点个数为N的问题中，需要分别以各路径顶点为源（Source），通过Dijkstra算法得到和其它节点的最小距离，总时间复杂度为$O(N \times V^2)$。随着总节点个数的增加，计算时间会非常大。

考虑到对各个节点进行Dijkstra的过程是独立的，可以将不同的节点交给不同的线程去完成。

并行化措施

相比于串行时循环地对每个路径上的节点使用Dijstra算法，我通过将不同的节点分配给不同的线程来达到提高执行效率的结果：做法是每次循环执行给定线程数的Dijkstra算法直到将路径上的所有点遍历完。

由于pthread_create参数只接受静态函数，我将函数内的Dijstra算法定义为静态成员函数，并将this指针和源节点id打包为结构体作为参数传入。

由于每个线程修改的变量都**分配在数组的不同位置**，理论上不存在资源竞争的可能，因此无需对操作进行加锁。

threadArgs C++ 复制代码

```cpp
// 作为参数传入线程函数的结构体
typedef struct threadArg
{
    public:
    threadArg(FixedSP*fixedSp, int source)
:fixedSp(fixedSp),source(source) {
    }
    FixedSP * fixedSp;
    int source;
} threadArg;
```

parallelization C++ 复制代码

```cpp

void FixedSP::multiDijkstra(int source, vector<int> intermediates){
    int threadNum = getThreadNum();
    pthread_t tid[threadNum];
    int pathLength = 1 + intermediates.size();
    int count = 0;
    vector<int> pathVec = intermediates;
    pathVec.insert(pathVec.begin(), source);

    while (count < pathLength){
        int threads_per_loop = min(pathLength - count, threadNum);
        for(int i = 0; i < threads_per_loop; i++){
            threadArg * arg = new threadArg (this,pathVec[count + i]);
            pthread_create(tid+i,nullptr,dijkstra_thread, arg);
        }
        for (int i = 0; i < threads_per_loop; ++i) {
            pthread_join(tid[i],nullptr);
        }
        count += threads_per_loop;
    }
}
```

性能测试

硬件环境

本次测试处理器为AMD Ryzen 7 4800U，8核。

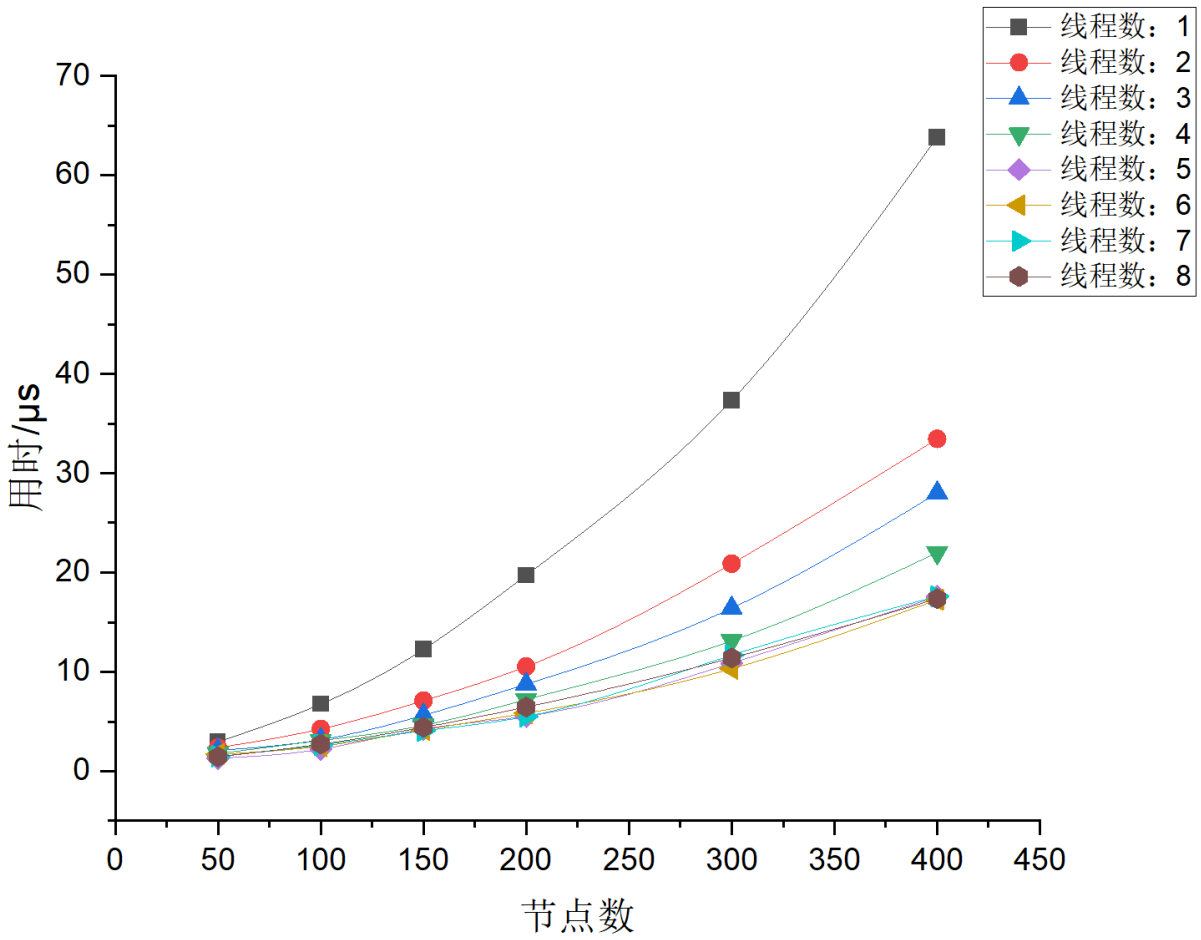
测试策略

通过改变线程（thread）个数，得到在无向邻接矩阵阶数（即总节点数）不同时，解决原问题所需要的时间变化。

> 注：整个问题的求解时间取决于Dijstra算法用时以及对节点进行全排列选出最短路径的用时，但本文只对Dijstra算法进行了并行化处理，因此以下的测试均是针对Dijkstra进行的，也即并行化措施中的**multiDijkstra函数**的性能。

测试图表

进程数\总节数	50	100	150	200	300	400
1	2.961	6.762	12.270	19.731	37.348	63.824
2	2.370	4.263	7.095	10.543	20.896	33.457
3	2.095	3.178	5.640	8.747	16.420	27.995
4	1.760	3.084	4.613	7.210	13.134	21.998
5	1.313	2.219	4.247	5.519	10.874	17.583
6	1.710	2.504	4.137	5.836	10.331	17.255
7	1.484	2.635	4.054	5.507	11.678	17.599
8	1.455	2.719	4.418	6.456	11.400	17.369



不同线程数下节点数对Dijkstra算法性能的影响图

实验现象与结论

现象

1. 由以上图表可知，线程数与必经节点数确定时，用时与总节点数大致满足O(V^2)的时间复杂度，符合Dijkstra算法的理论时间复杂度；
2. 改变使用线程数，可以看到性能有明显的提升，从单线程变为双线程后，用时变为一半，之后直到线程为5，性能提升虽达不到线性，但也有较大提升，在线程数达到5以后，提升线程数对性能几乎没有影响。

结论

实验的结论基本达到了预期，与理论也较吻合：增加线程数，可以提升代码执行效率，在有些时候这种提升是接近线性的，但有时提升并不明显乃至几乎没有影响。

在本实验中，线程数达到到5之后性能没有明显变化，推测是由于我在设计测试集时将路径长度固定为10，每次循环执行线程个数的Dijkstra线程，因此当线程达到5以上时，只要线程数小于10，都需要2次循环来完成所有的待做线程，而线程的创建和销毁需要一定时间，因此进程的提升对性能提升不大。本次测试所用的处理器为8核，如果将路径长度增加到16以上，应当可以看见线程从1–8性能的持续增加，然而路径过长会导致进行对路径排列组合时的开销太大，这则是另一个需要讨论的问题了。

综上所述，通过将代码并行化，可以有效地提高性能，本质上是将总的计算量分给多个线程去共同完成来提高这部分的执行效率。在并行过程中，必须谨慎地处理全局变量/静态变量，避免各线程之间相互影响，如有必要，可对部分操作进行加锁处理。