Dijkstra算法

基础版 dijkstra算法

基于邻接表存图

```
const int N = 1e7 + 10;

#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

int n, m;
int h[N], e[N], w[N], ne[N], idx;
int dist[N];
bool st[N];

void add(int a,int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}

int dijsktra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for(int i=0;i<n-1;i++){
        int t=-1;

#pragma omp parallel for
        for(int j=1;j<=n;j++){
        register int *k = dist;
            if(!st[j] && (t==-1 || *(k+t) > *(k+j)))
                t = j;
        }

/*
        for(int j=1;j<=n;j++){
            if(!st[j] && (t==-1 || dist[t] > dist[j]))
                t = j;
        }
*/

        for(int j=h[t]; j!=-1; j=ne[j]){
            int k = e[j];
            if(unlikely(dist[k] > dist[t]+w[j])){
                dist[k] = dist[t]+w[j];
            }
        }
        st[t] = true;
    }
```

```
    if(dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

Baseline

```
chen@chen-virtual-machine:~/dijkstra$ g++ dijkstra.cpp -fopenmp  -o dijkstra
chen@chen-virtual-machine:~/dijkstra$ ./dijkstra
ANSWER is 25000179
Dijkstra运行耗时: 25407.171800 ms
```

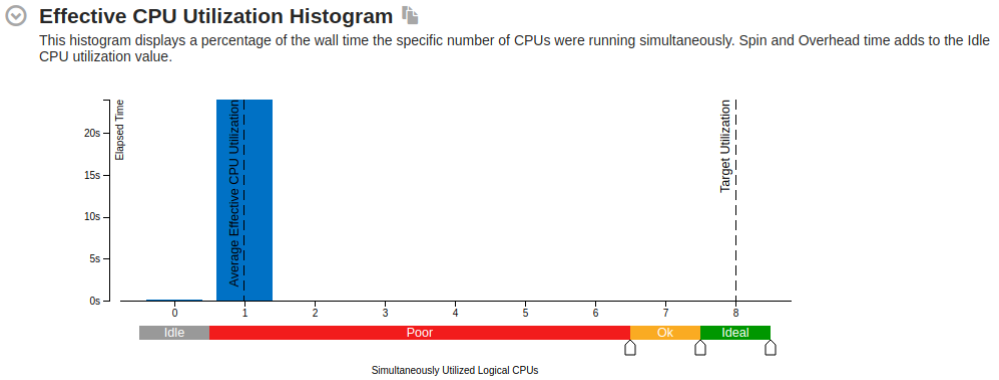
编译选项

```
chen@chen-virtual-machine:~/dijkstra$ g++ dijkstra.cpp -O3 -ffast-math -funroll-all-loops -mavx -mtune=native -fopenmp  -o dijkstra
```

```
-O3 -ffast-math -funroll-all-loops -mavx -mtune=native -fopenmp
```

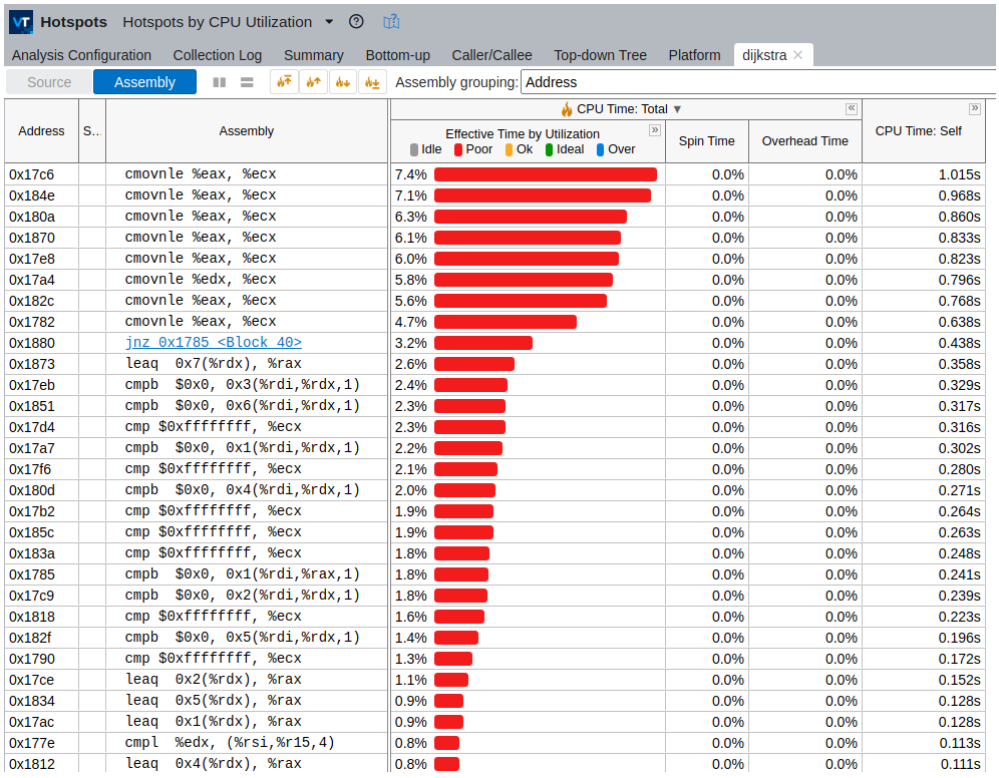
```
chen@chen-virtual-machine:~/dijkstra$ ./dijkstra
ANSWER is 25000179
Dijkstra运行耗时: 13348.390234 ms
```

热点分析



Average Effective CPU Utilization: 0.999

Target Utilization: 8

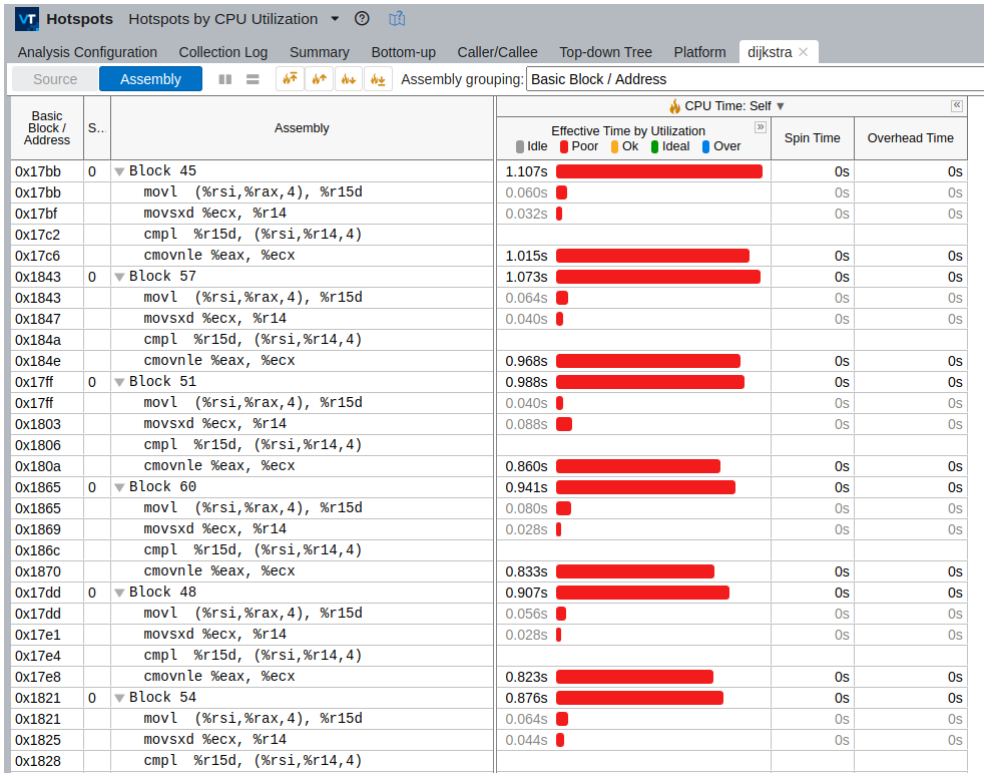


Hotspots Hotspots by CPU Utilization

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform | dijkstra

Source | Assembly | Assembly grouping: Address

Address	S..	Assembly	CPU Time: Total ▾ — Effective Time by Utilization (Idle Poor Ok Ideal Over)	Spin Time	Overhead Time	CPU Time: Self
0x17c6		cmovnle %eax, %ecx	7.4%	0.0%	0.0%	1.015s
0x184e		cmovnle %eax, %ecx	7.1%	0.0%	0.0%	0.968s
0x180a		cmovnle %eax, %ecx	6.3%	0.0%	0.0%	0.860s
0x1870		cmovnle %eax, %ecx	6.1%	0.0%	0.0%	0.833s
0x17e8		cmovnle %eax, %ecx	6.0%	0.0%	0.0%	0.823s
0x17a4		cmovnle %edx, %ecx	5.8%	0.0%	0.0%	0.796s
0x182c		cmovnle %eax, %ecx	5.6%	0.0%	0.0%	0.768s
0x1782		cmovnle %eax, %ecx	4.7%	0.0%	0.0%	0.638s
0x1880		jnz 0x1785 <Block 40>	3.2%	0.0%	0.0%	0.438s
0x1873		leaq 0x7(%rdx), %rax	2.6%	0.0%	0.0%	0.358s
0x17eb		cmpb $0x0, 0x3(%rdi,%rdx,1)	2.4%	0.0%	0.0%	0.329s
0x1851		cmpb $0x0, 0x6(%rdi,%rdx,1)	2.3%	0.0%	0.0%	0.317s
0x17d4		cmp $0xffffffff, %ecx	2.3%	0.0%	0.0%	0.316s
0x17a7		cmpb $0x0, 0x1(%rdi,%rdx,1)	2.2%	0.0%	0.0%	0.302s
0x17f6		cmp $0xffffffff, %ecx	2.1%	0.0%	0.0%	0.280s
0x180d		cmpb $0x0, 0x4(%rdi,%rdx,1)	2.0%	0.0%	0.0%	0.271s
0x17b2		cmp $0xffffffff, %ecx	1.9%	0.0%	0.0%	0.264s
0x185c		cmp $0xffffffff, %ecx	1.9%	0.0%	0.0%	0.263s
0x183a		cmp $0xffffffff, %ecx	1.8%	0.0%	0.0%	0.248s
0x1785		cmpb $0x0, 0x1(%rdi,%rax,1)	1.8%	0.0%	0.0%	0.241s
0x17c9		cmpb $0x0, 0x2(%rdi,%rdx,1)	1.8%	0.0%	0.0%	0.239s
0x1818		cmp $0xffffffff, %ecx	1.6%	0.0%	0.0%	0.223s
0x182f		cmpb $0x0, 0x5(%rdi,%rdx,1)	1.4%	0.0%	0.0%	0.196s
0x1790		cmp $0xffffffff, %ecx	1.3%	0.0%	0.0%	0.172s
0x17ce		leaq 0x2(%rdx), %rax	1.1%	0.0%	0.0%	0.152s
0x1834		leaq 0x5(%rdx), %rax	0.9%	0.0%	0.0%	0.128s
0x17ac		leaq 0x1(%rdx), %rax	0.9%	0.0%	0.0%	0.128s
0x177e		cmpl %edx, (%rsi,%r15,4)	0.8%	0.0%	0.0%	0.113s
0x1812		leaq 0x4(%rdx), %rax	0.8%	0.0%	0.0%	0.111s



Hotspots Hotspots by CPU Utilization

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform | dijkstra

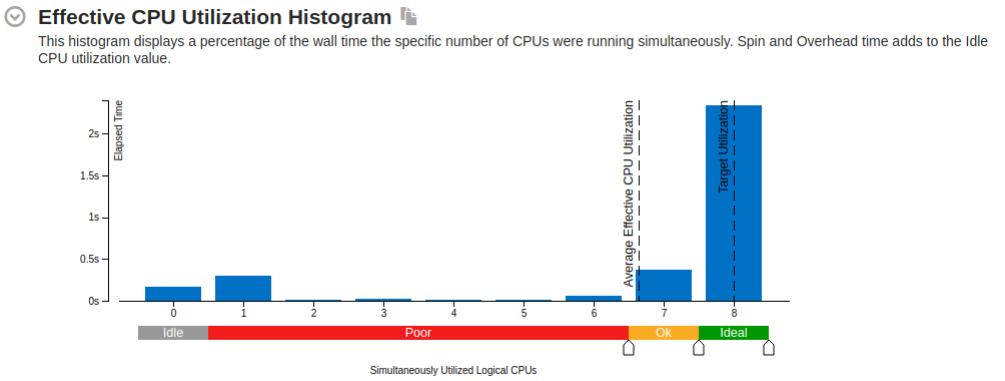
Source | Assembly | Assembly grouping: Basic Block / Address

Basic Block / Address	S..	Assembly	CPU Time: Self ▾ — Effective Time by Utilization (Idle Poor Ok Ideal Over)	Spin Time	Overhead Time
0x17bb	0	▾ Block 45	1.107s	0s	0s
0x17bb		movl (%rsi,%rax,4), %r15d	0.060s	0s	0s
0x17bf		movsxd %ecx, %r14	0.032s	0s	0s
0x17c2		cmpl %r15d, (%rsi,%r14,4)			
0x17c6		cmovnle %eax, %ecx	1.015s	0s	0s
0x1843	0	▾ Block 57	1.073s	0s	0s
0x1843		movl (%rsi,%rax,4), %r15d	0.064s	0s	0s
0x1847		movsxd %ecx, %r14	0.040s	0s	0s
0x184a		cmpl %r15d, (%rsi,%r14,4)			
0x184e		cmovnle %eax, %ecx	0.968s	0s	0s
0x17ff	0	▾ Block 51	0.988s	0s	0s
0x17ff		movl (%rsi,%rax,4), %r15d	0.040s	0s	0s
0x1803		movsxd %ecx, %r14	0.088s	0s	0s
0x1806		cmpl %r15d, (%rsi,%r14,4)			
0x180a		cmovnle %eax, %ecx	0.860s	0s	0s
0x1865	0	▾ Block 60	0.941s	0s	0s
0x1865		movl (%rsi,%rax,4), %r15d	0.080s	0s	0s
0x1869		movsxd %ecx, %r14	0.028s	0s	0s
0x186c		cmpl %r15d, (%rsi,%r14,4)			
0x1870		cmovnle %eax, %ecx	0.833s	0s	0s
0x17dd	0	▾ Block 48	0.907s	0s	0s
0x17dd		movl (%rsi,%rax,4), %r15d	0.056s	0s	0s
0x17e1		movsxd %ecx, %r14	0.028s	0s	0s
0x17e4		cmpl %r15d, (%rsi,%r14,4)			
0x17e8		cmovnle %eax, %ecx	0.823s	0s	0s
0x1821	0	▾ Block 54	0.876s	0s	0s
0x1821		movl (%rsi,%rax,4), %r15d	0.064s	0s	0s
0x1825		movsxd %ecx, %r14	0.044s	0s	0s
0x1828		cmpl %r15d, (%rsi,%r14,4)			

并行化

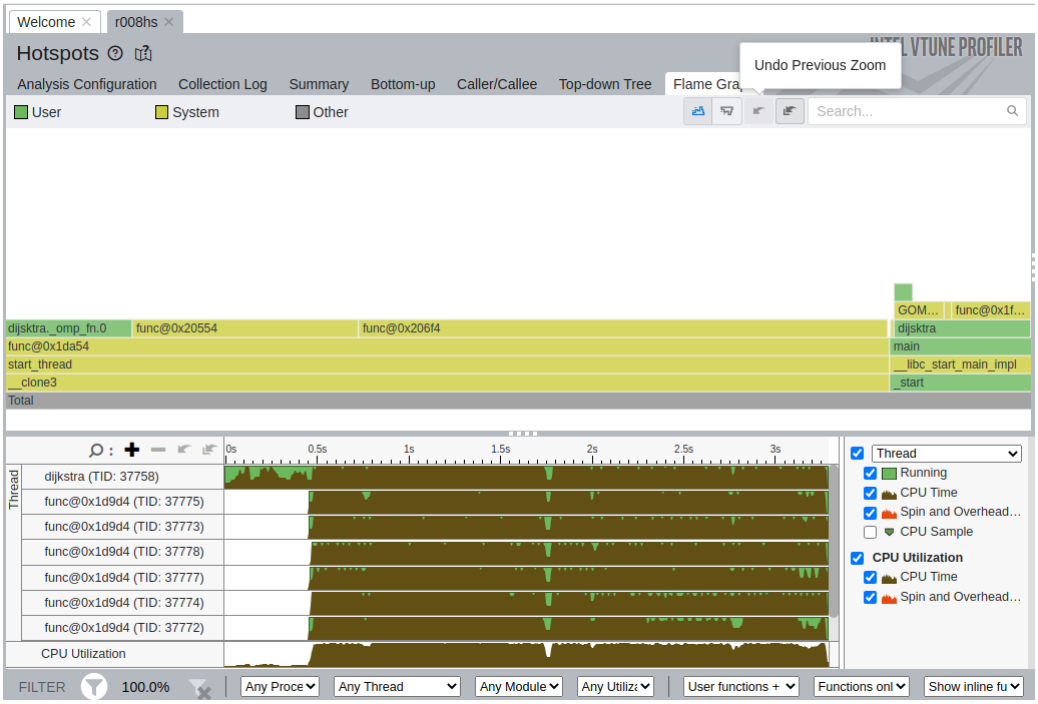
```
30 #pragma omp parallel for
31         for(int j=1;j<=n;j++){
32         register int *k = dist;
33             if(!st[j] && (t==-1 || *(k+t) > *(k+j)))
34                 t = j;
35         }
```

```
chen@chen-virtual-machine:~/dijkstra$ ./dijkstra
ANSWER is 25000179
Dijkstra运行耗时: 2173.732957 ms
```



Average Effective CPU Utilization: 6.648　（平均有效CPU利用率:6.648）

Target Utilization: 8　（目标利用率:8）



效果还阔以

堆优化版 dijkstra算法

```
const int N = 1e7 + 10;

typedef pair<int, int> PII;
int n, m;
int h[N], e[N], w[N], ne[N], idx;
int dist[N];
bool st[N];

void add(int a,int b, int c)
{
```

```
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}

int dijsktra()
{
    memset(dist, 0x3f, sizeof dist);
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    dist[1] = 0;
    heap.push({0, 1});

    while(heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int x = t.second, distance = t.first;

        if(st[x]) continue;
        st[x] = true;

        for(int i = h[x]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > dist[x] + w[i])
            {
                dist[j] = dist[x] + w[i];
                heap.push({dist[j], j});
            }
        }
    }

    if(dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

Baseline

```
chen@chen-virtual-machine:~/dijkstra$ ./dijkstra2
25000179
Dijkstra运行耗时: 403.877146 ms
```

编译选项

```
-O3 -ffast-math -funroll-all-loops -mavx -mtune=native -fopenmp
```

```
chen@chen-virtual-machine:~/dijkstra$ ./dijkstra2
25000179
Dijkstra运行耗时: 59.327957 ms
```

VTune

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
__memset_avx2_unaligned_erms	libc.so.6	1.020s	79.7%
__isoc99_fscanf	libc.so.6	0.116s	9.1%
main	dijkstra2	0.052s	4.1%
std::__adjust_heap<__gnu_cxx::__normal_iterator<std::pair<int, int>*, std::vector<std::pair<int, int>, std::allocator<std::pair<int, int>>>>, long, std::pair<int, int>, __gnu_cxx::__ops::_Iter_comp_iter<std::greater<std::pair<int, int>>>>	dijkstra2	0.052s	4.1%
dijsktra	dijkstra2	0.020s	1.6%
[Others]	N/A*	0.020s	1.6%

*N/A is applied to non-summable metrics.

spfa最短路 (队列)

```
const int N = 1e7 + 10;

int n, m;
int dist[N];
int h[N], e[N], ne[N], w[N], idx;
bool st[N];

void add(int a,int b, int c){
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}

int spfa(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;

    while (q.size()){
        int t = q.front();
        q.pop();
        st[t] = false;
        for (int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            if (dist[j] > dist[t] + w[i]){
                dist[j] = dist[t] + w[i];
                if (!st[j]){
                    q.push(j);
                    st[j] = true;
                }
            }
        }
```

```
    }
    return dist[n];
}
```

Baseline

```
chen@chen-virtual-machine:~/dijkstra$ g++ spfa.cpp -fopenmp -o spfa
chen@chen-virtual-machine:~/dijkstra$ ./spfa
spfa运行耗时: 119671.488236 ms
25000179
```

编译选项

```
-O3 -ffast-math -funroll-all-loops -mavx -mtune=native -fopenmp
```

```
chen@chen-virtual-machine:~/dijkstra$ ./spfa
spfa运行耗时: 45487.688959 ms
25000179
```

并行化Dijkstra算法

传统串行 Dijkstra 算法

- 采用广度优先搜索思想，它的主要特点是选定起始点后，一个点一个点地求取最短距离，通过邻点逐步扩展，不断更新，直至求出起始点到目标点的最短距离后才停止。
- 假定赋权图G（V，E，W），V、E、W分别为图的顶点集、图的边集和图的边权集。假定起点为u0，设集合 S ∈ V ，初始时集合 S 中仅包含起点 u0。对于任意一个属于顶点集 V-S 的顶点 u1，若已知起点 u0 至该顶点u1 的最短距离，则将该顶点u1放入集合S中，并记录起点u0到该顶点u1的最短距离。
- 在集合 V-S 中找到距离起点 u0 最近的顶点，记为顶点q，将顶点q加入集合S的同时将顶点q从集合V-S中移除。然后将顶点 q 视为新的起点，重复之前的步骤，直至目标点被加入集合 S 中。

Dijkstra 最短路径算法本身存在并行化的不足

- 集合 S （标记点集合） 每次循环迭代之后定点个数都会加 1，每次迭代都依赖于上次迭代的结果，循环之间存在依赖关系

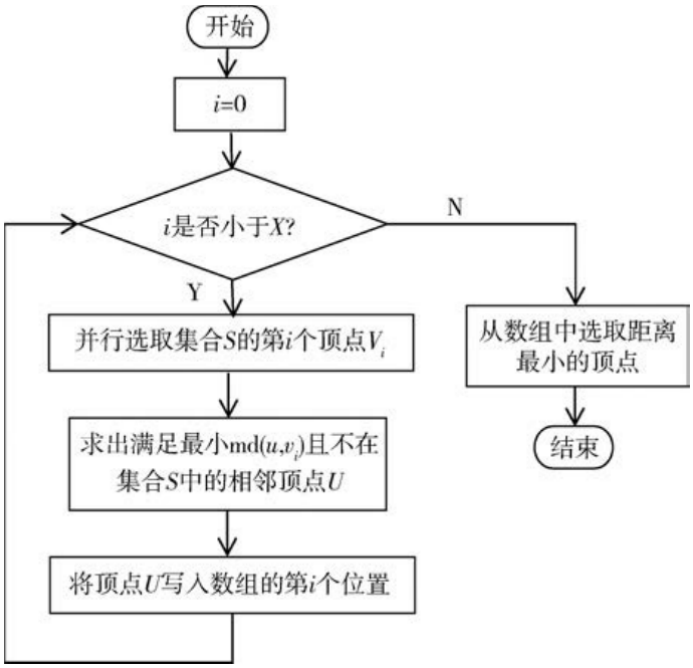
选择可并行化的主要区域

```
int t = -1;
for(int j = 1; j <= n; j ++)
    if(!st[j] && (t == -1 || dist[t] > dist[j]))
        t = j;
```

- 选择未标注节点中的最小节点：未标记节点以无序的形式存放在一个数组或一个链表内，每次选择最短路径节点都必须把所有未标记节点扫描一遍，当节点数目较大时，这将成为制约计算速度的关键因素。

并行算法（一）

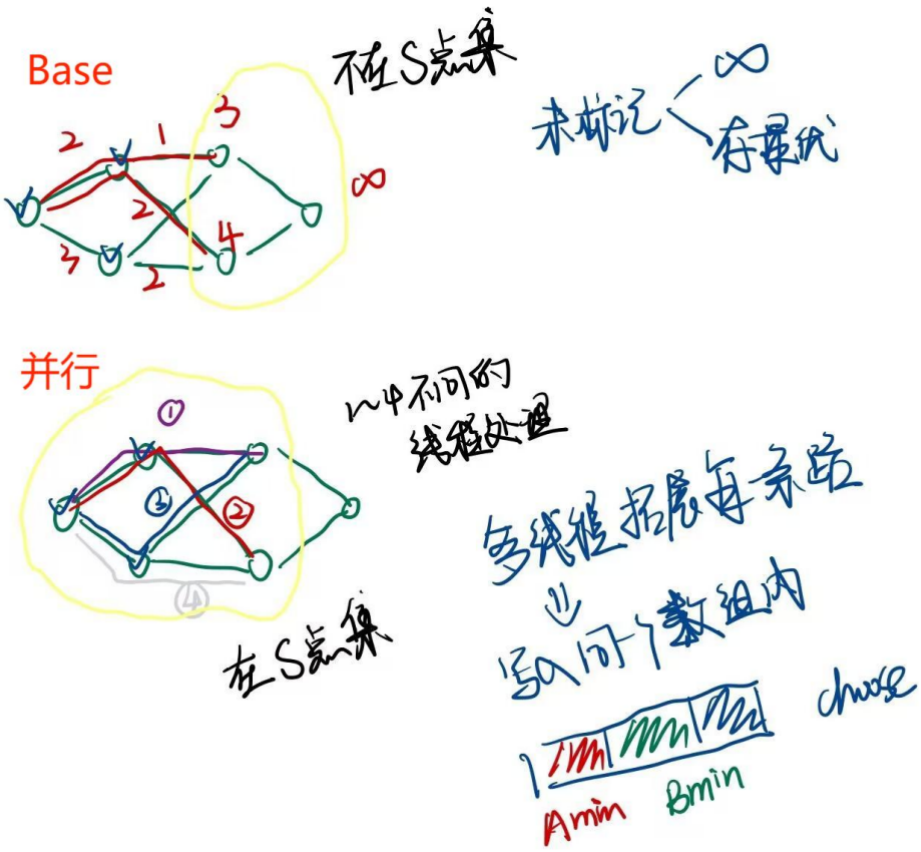
每个线程计算一个顶点的所有边，从中取得最小边并保存在一个数组的不同位置，然后从数组中找出最小的值，得到最近距离的一个顶点

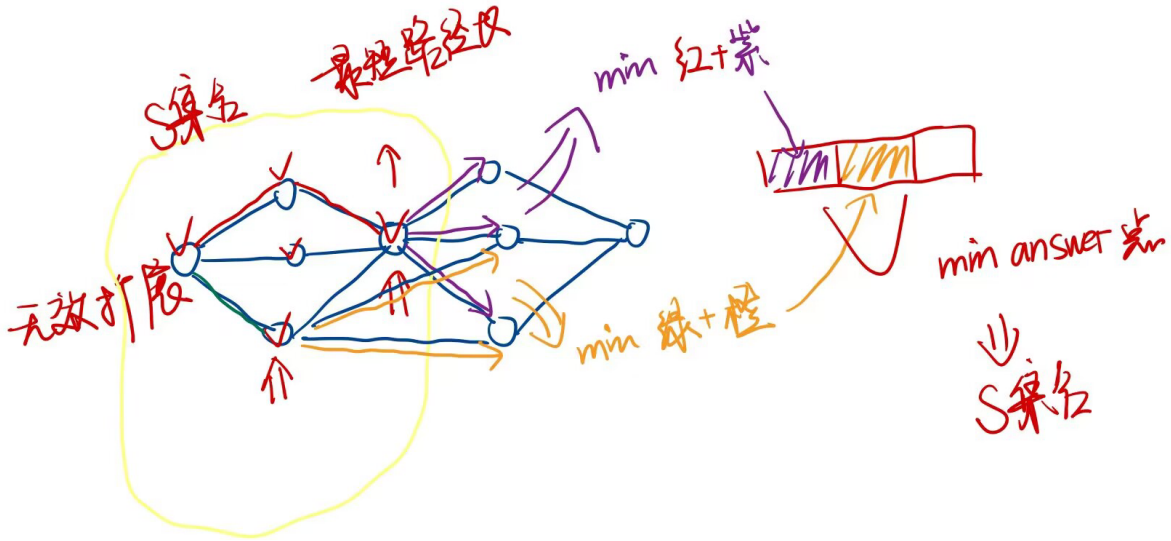


$$md(ui,\ vi) = w(ui,\ vj) + Len(vj)$$

利用多线程，直接对于**所有的集合S**的点进行求取md处理

跟基础的Dijkstra有点不一样





并行算法（二）

两个线程的并行化

采用**双线程**分别从起点和终点同时按照 Dijkstra 算法最短路径的搜索分析，采用节点标记法动态地进行子网分割，最终将两个子网的搜索结果进行拼接汇总，获取最短路径。

思想：

图 1　串行思路

Fig. 1　Serialization

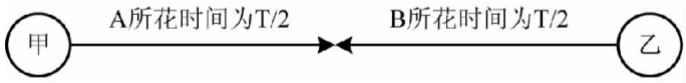
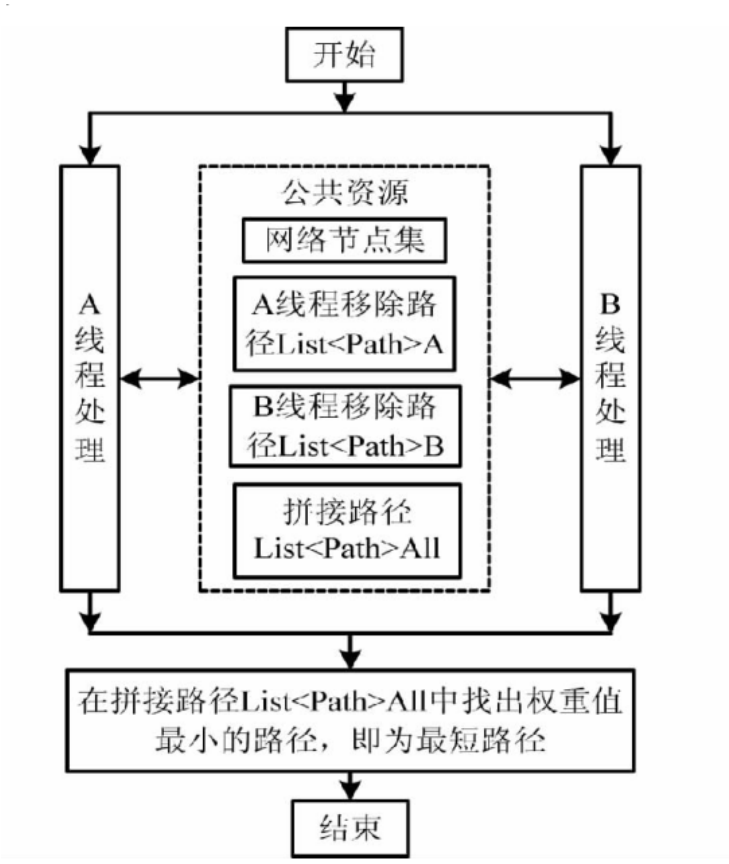


图 2　并行思路

最后根据拓扑关系进行路径集合的拼接，得出最短路径



优点：

易于线程间的协调控制和数据通信，两个线程在处理过程中根据节点的状态标记，动态地进行两个子网的分割，一个线程处理慢了，另一个线程可以进行“帮忙”，直到处理完毕，两个子网才划分结束

软硬件的要求低

资源利用率高

缺点：

并行化程度不高

边界处理较为复杂

并行算法（三）

定义：

将 L 定义为起点 u0 到另一个顶点 u1 的**最短**路径权值，若顶点 u1 在第 r 步获得了标号 L，则称顶点 u1 在第
r 步获得永久性标号，r ≥ 0

将 l 定义为起点u0到顶点u1的最短路径权值**上限。**若顶点 u1 在第 r 步可以达到 l，则称 u1 在第 r 步获得了临时
性标号，r ≥ 0

理论：

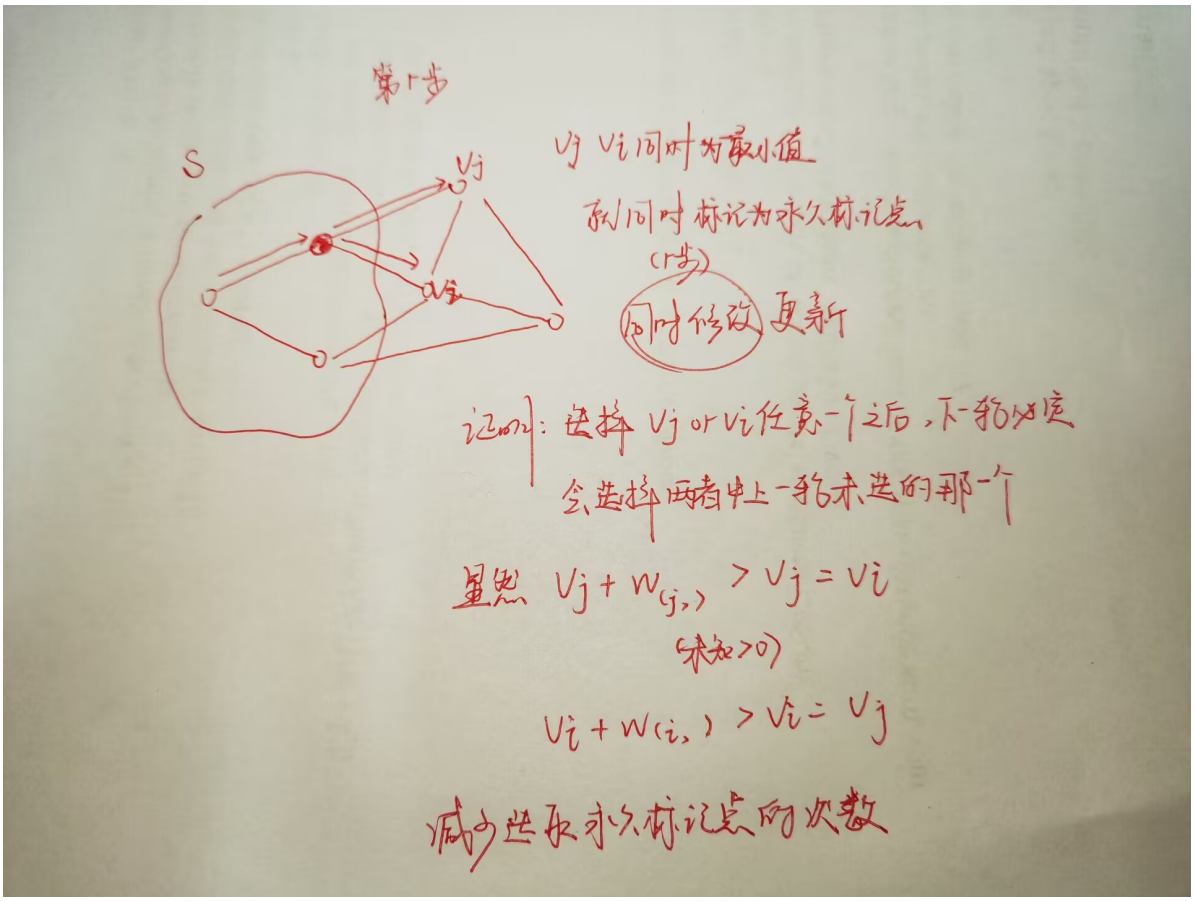
在选择永久性标号的过程中，如果在第 r 步时，有多个顶点已经获得最小临时性标号，那么应该同时赋予这些顶点永久性标号

目的：

明显减少对临时性标号集合的遍历次数，达到缩短计算时间，提升计算效率的目的；同时也是并行化的理想对象

可以尽可能地减少永久性标号选择次数，进而缩短计算时间，提升算法运行效率

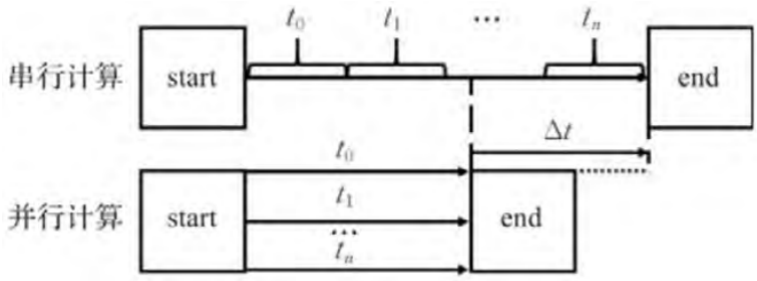
定理1 Dijkstra标号法中，在同一步将多个拥有最小标号 l 的顶点标记上标号 L，最短路的计算结果不变。



并行多标号Dijkstra 算法的时间复杂度:

$$O（h（n/k+logn））$$

核心采用**分治法**：将复杂问题等效转化为多个等价的较简单的子问题



> ps.　OpenMP耗时：
>
> critical 指令实现枷锁效果：使用锁来实现互斥：原子操作 = 7：3.5：1

由于需要修改获得永久性标号 L 顶点的邻点，导致多标号的Dijkstra需要修改的点数增加

储存临时标号 l 的数组作为此并行算法的**共享数据**

分配各个线程的工作量，让**线程固定处理若干个顶点**的标号 l 值，从而做到**数据独立**

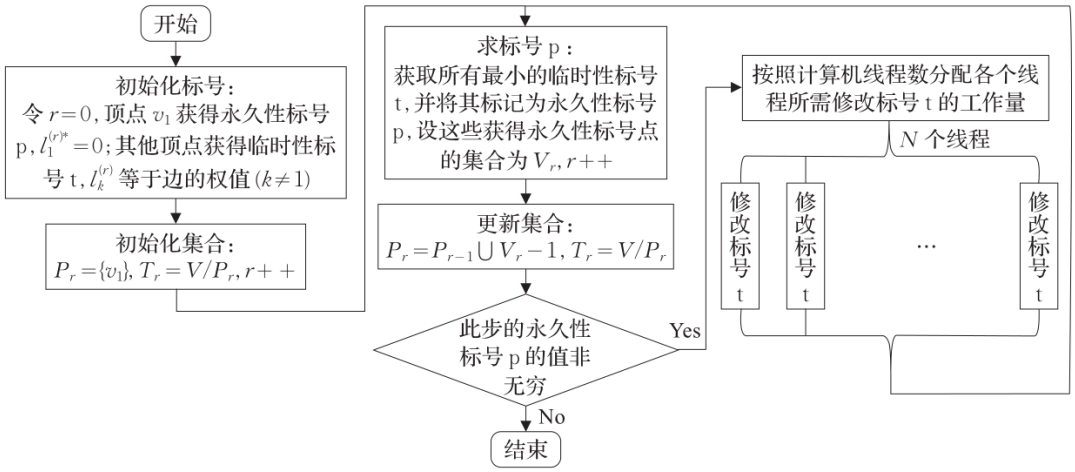


图4　多标号 Dijkstra 并行算法流程图