关于Dijkstra算法的OpenMP实现

17342006 丁聿宁

作业内容：

问题简述：

针对经典的Dijkstra算法的串行实现，设计其共享内存并行机制，分析优化的思路和流程并使用OpenMP实现其并行化，最终给出实验结果，包括并行结果正确性说明，2/4/8/16线程并行执行的时间（Wall Time）、加速比（并行时间/串行时间）和效率（加速比/线程数）。

实验平台：

超算习堂

并行代码设计：

`minDistance`函数

`minDistance`函数中的循环存在数据依赖，要分支语句要判断min，然后要对min做修改。为了避免数据依赖，我设置了my_min以及idx数组，来记录每个核各自计算的最小值，然后在遍历my_min以及idx数组来找出dist数组里面的最小值的索引。

修改如下：

```
int minDistance(int dist[], bool sptSet[], int thread_count)
{
// Initialize min value
int min = INT_MAX, min_index;
int v;
int my_min[thread_count];
int idx[thread_count];
//初始化my_min数组的值，不用parallel for是因为这个循环次数很小
for(v = 0; v < thread_count; v++){
    my_min[v] = INT_MAX;
}
# pragma omp parallel for num_threads(thread_count)
for (v = 0; v < V; v++){
    int id = omp_get_thread_num();
    if (sptSet[v] == false && dist[v] <= my_min[id]){
        my_min[id] = dist[v];
        idx[id] = v;
    }
}
//找出总的最小值
for(v=0;v<thread_count;v++){
    if(min>=my_min[v]){
        min=my_min[v];
```

```
            min_index=idx[v];
        }
    }

    return min_index;
    }
```

`printSolution`函数

`printSolution`函数不用做修改，虽然它含有一个循环，但是这个循环是为了按顺序输出每个点距顶点0的最短距离的，如果用`parallel for`优化可能会导致输出顺序不一致，

`dijkstra`函数

`dijkstra`函数有一个简单的循环，还有一个嵌套循环。对于简单循环，它没有数据依赖，且for语句可并行化的，因此可以直接使用`parallel for`。观察嵌套循环，不存在数据依赖（graph[u] [u] = 0，不会出现将dist[u] 修改从而导致其他数据发生改变的情况），而且for是可并行化的，因此可以使用`parallel for`。而对于嵌套循环，上网查阅资料，了解到一个原则是应该尽量少使用`parallel for`，因为`parallel for`也需要时间开销。所以我在内层循环上加了`parallel for`。我同时也测试了将`parallel for`放在外层循环，以及放在内外层循环中，多次运行发现这两种情况得到的输出结果不少是错误的，即某些dist的值变得很大。而对于将`parallel for`放在内层循环的情况，多次运行得到的输出结果几乎都是正确的，这其中的原因并不是很了解。

最终`dijkstra`函数修改如下：

```
void dijkstra(int graph[V][V], int src, int thread_count)
{
    int dist[V];     // The output array. dist[i] will hold the shortest
                     // distance from src to i

    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                     // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    int i;
# pragma omp parallel for num_threads(thread_count)
    for (i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    int count;
    for (count = 0; count < V-1; count++)
    {
    // Pick the minimum distance vertex from the set of vertices not
    // yet processed. u is always equal to src in the first iteration.
    int u = minDistance(dist, sptSet, thread_count);

    // Mark the picked vertex as processed
    sptSet[u] = true;
```

```
    // Update dist value of the adjacent vertices of the picked vertex.
    int v;
# pragma omp parallel for num_threads(thread_count)
    for (v = 0; v < V; v++)
        // Update dist[v] only if is not in sptSet, there is an edge from
        // u to v, and total weight of path from src to v through u is
        // smaller than current value of dist[v]
        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                                    && dist[u]+graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}
```

最终输出结果：

```
PS C:\Users\28562\Desktop\高性能> ./dijkstra.exe 4
Vertex Distance from Source
0         0
1         4
2         12
3         19
4         21
5         11
6         9
7         8
8         14
running time: 13
```

而串行输出结果：

> Vertex Distance from Source
>
> 0 0 1 4 2 12 3 19 4 21 5 11 6 9 7 8 8 14

可见与串行代码输出结果一致。

实验结果

接下来测试时间，首先修改主函数：

```
int main(int argc, char* argv[])
{
/* Let us create the example graph discussed above */
```

```
    int thread_count = strtol(argv[1], NULL, 10);

    //创建大矩阵
    int **graph;
    int i, j;
    graph=(int**)malloc(sizeof(int*)*V);
    for(i=0;i<V;i++)
    graph[i]=(int*)malloc(sizeof(int)*V);
//  srand((unsigned)time(NULL));

    //给大矩阵赋值，它是对称阵且对角线元素为0
    for (i = 0; i < V; i++)
        for (j = i + 1; j < V; j++)
            graph[i][j] = rand() % 100;
    for (i = V - 1; i >= 0; i--)
        for (j = 0; j < i; j++)
            graph[i][j] = graph[j][i];
    for (i = 0; i < V; i++)
        graph[i][i] = 0;

    //记录运行时间
    double start = omp_get_wtime();
    dijkstra(graph, 0, thread_count);
    double finish = omp_get_wtime();
    printf("time: %f\n", (finish - start));

    //删除矩阵
    for (i = 0; i < V; i++)
        free(graph[i]);
    free(graph);
    return 0;
}
```

这里可以用宏来控制V的值，从而控制测试矩阵的大小。我将V设为9999，使测试矩阵尽可能大，从而体现并行程序的优越性。

串行程序运行10次的运行时间（线程数为1时）：

1.979938 1.969012 1.978378 2.109737 1.968769 1.964456 1.967034 1.974388 1.971236 1.974924

avg: 1.985787

2线程运行10次的运行时间：

1.810476 1.164868 1.606288 1.706605 1.158684 1.516517 1.672033 1.225980 1.399542 1.227768

avg: 1.448876

4线程运行10次的运行时间：

1.195678 1.280589 1.066110 0.777361 2.282031 1.112403 1.475792 1.066292 1.160471 1.550599

avg: 1.296733

8线程运行10次的运行时间：

1.740586 1.146767 1.273899 1.921708 2.185295 1.859755 2.081436 1.752513 1.415328 1.262611

avg: 1.66399

16线程运行10次的运行时间：

2.613080 2.746722 2.895709 2.616870 2.700955 2.615063 2.554249 1.977801 2.864576 2.667949

avg: 2.625297

可得以下表格：

线程数	**1**	**2**	**4**	**8**	**16**
平均运行时间	1.985787	1.448876	1.296733	1.663990	2.625297
加速比	---	1.370571	1.531377	1.193389	0.756405
效率	---	0.685286	0.382844	0.149174	0.047275

分析表格，可知并行程序加速比在2线程到4线程有提升，但在8线程之后就减少了，16线程的运行速度甚至比串行程序慢。由于加速效果不明显，效率自然是随着线程数的增加而减少。我认为并行程序加速比与效率不高的原因在于在`dijkstra`函数中的嵌套循环只是优化了内层循环，查阅资料可知，对于嵌套循环，在内外层循环循环次数相差不大的情况下，优化外层循环要比优化内层循环的效果更显著，优化内层循环对程序的效率提升实际不大。但是这里由于存在数据依赖的关系，是不能优化外层循环的。与此同时，申请线程以及parallel for也有开销，在8线程、16线程的情况下，优化的程度不高，但开销很大，这就导致了出现加速比减小的情况。

（注意，以下的不是修改代码，而是验证想法）

为了验证我的想法，我去掉了内层循环的`parallel for`，而在外层循环添加parallel for，得到结果如下：

线程数	**1**	**2**	**4**	**8**	**16**
平均运行时间	1.985787	1.144495	0.603315	0.323870	0.212747
加速比	---	1.735077	3.291457	6.131432	9.334031
效率	---	0.867539	0.822865	0.766429	0.583377

如表格所示，得到了我们预期的并行程序的优化效果。

但是很可惜的是，由于存在函数依赖，我们是不能在外层循环上加入`parallel for`的，因此实际上只能得到表格1的效果。