并行大作业报告

实验简介

实现思路

测试性能

可以改进的地方

1700012929 季陆炀

实验简介

对于一棵有N个节点的树T，根节点编号为0，自树根至树叶的编号呈深度优先次序。每个结点具有以下六种属性(括号内为缩写):index(id)，upper(u)、lower(l)、diagonal(d)、 right-side-hand(rhs)、parent(p)，其中 parent 代表的父结点的 id。

该问题包含两个阶段:第一阶段完成由树叶至树根的一次遍历，记作 backward sweep; 第二阶段完成由树根至树叶的一次遍历，记作 forward sweep，两个阶段的次序不可交换。算法的伪代码如下：

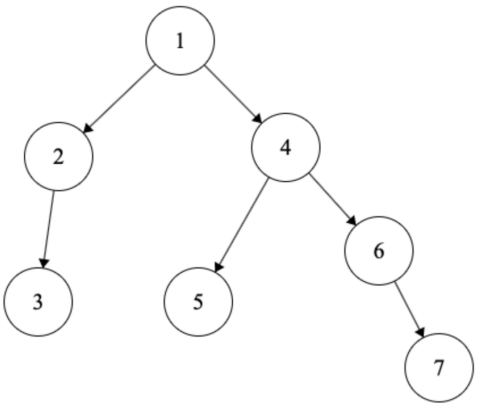
```
void HinesAlgo(double *u, double *l, double *d, double *rhs, int* p, int N){
    int i;
    double factor;
    //Backward Sweep
    for i = N-1 to 1 do
        factor = u[i] / d[i];
        d[p[i]] -= factor * l[i];
        rhs[p[i]] -= factor * rhs[i];
    rhs[0] /= d[0];
    //Forward Sweep
    for i = 1 to N-1 do
        rhs[i] -= l[i] * rhs[p[i]];
        rhs[i] /= d[i];
}
```

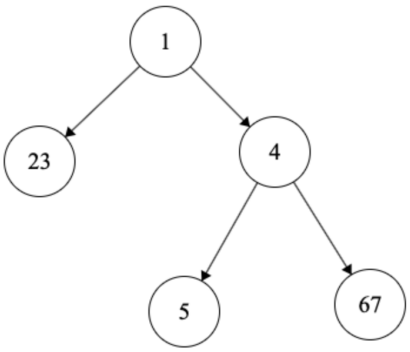
要求编写Hines的串行版本和并行版本，对比时间、加速比，并将结果输出到文件，共N行，按index升序输出index upper lower right-side-hand diagonal 用空格隔开。只有时间会输出到命令行，

简单分析一下两次遍历的过程：分为两部分，第一部分自下而上，每次用子节点计算出一个factor值，然后去更新父节点的diagonal和rhs值。第二部分自上而下，每次用父节点的rhs去更新子节点的rhs。

实现思路

为了实现Hines算法的并行版本，首先需要找到可以并行的子树部分。我们知道如果节点之间存在父子关系，那这两个节点是必须不能并行的，因为节点之间存在着数据依赖。所以产生了如下一个思路：对于一串单链形状的子树，我们把他们组合在一起，作为一个task，这个task是不可分割，必须串行的。合并完task之后，所有的task又形成了一个新的子树，如下图所示，左边的23和67节点都可以合并作为一个task。





在新的task树中，不难发现，同层(相同深度)的任务节点是可以并行的，因为不存在数据依赖。但要注意，仍然存在一种冲突，就是在backward sweep中，是子节点更新父节点，比如上图的5和67，可能同时对4进行更新，所以开始采取的是这样的策略：

```
for(int i=depth;i>=0;i--){
#pragma omp parallel for num_threads(THREAD_NUM) schedule(dynamic)
            for(int j=0;j<level[i].size();j++){
                task* tmptask=level[i][j];
                for(int k=tmptask->nodes.size()-1;k>=0;k--){
                    int now=tmptask->nodes[k];
                    if(now){
                        double factor=upper[now]/diagonal[now];
                        double lfactor=factor*lower[now];
                        double rhsfactor=factor*rhs[now];
                        int pnow=parent[now];
                        #pragma omp atomic
                            diagonal[pnow]-=lfactor;
                        #pragma omp atomic
                            rhs[pnow]-=rhsfactor;
                    }
```

```
                    else{
                        rhs[0]/=diagonal[0];
                    }
                }
            }
    }
```

其中level[i]是bfs得到的第i层所有的task，然后对于原先更新父节点的表达式，我们尽可能并行地算出factor*l[i]和factor*rhs[i]等数据，最后在更新父节点的时候给他一个atomic声明，确保不会出现冲突并且尽可能的并行。另外在forward sweep中，由于是用父节点更新子节点，所以完全不会存在冲突。
但其实在backward sweep的过程中还可以进行优化。出现冲突的根本原因是因为，不同节点可能有同一个父亲，而不同父节点的子节点一定是不同的，所以用父节点更新子节点的方法是不存在冲突的。所以我们可以把backward sweep的扫描也变成由父节点更新子节点的方式：

```
for(int i=depth;i>=0;i--){
#pragma omp parallel for num_threads(THREAD_NUM) schedule(dynamic)
            for(int j=0;j<level[i].size();j++){
                task* tmptask=level[i][j];
                for(int k=tmptask->nodes.size()-1;k>=0;k--){
                    int now=tmptask->nodes[k];
                    for(int kk=0;kk<childs[now].size();kk++){
                        int child=childs[now][kk];
                        double factor=upper[child]/diagonal[child];
                        diagonal[now]-=factor*lower[child];
                        rhs[now]-=factor*rhs[child];
                    }
                }
            }
    }
```

预处理childs数组，记录节点的所有孩子列表。然后对于每一个节点，扫描其所有子节点，用子节点的l[child]和rhs[child]来更新父节点，这样就避免了冲突。前后速度进行对比，后者略有提升，尤其是在数据量不是很大的情况下。另外，调度策略采用的是dynamic，经过测试，采用dynamic调度策略可以比默认调度要略快一些。

测试性能

在服务器上对不同case进行串行和并行的性能测试，采用thread_num=20，仅记录hines算法两段扫描所需的时间，运行时间取三次平均值，结果如下：

case	serial	parallel	加速比
1	0.000151s	0.000854s	0.176815
2	0.000160s	0.000580s	0.275862
3	0.000165s	0.000907s	0.181918
4	0.000160s	0.001341s	0.119313
5	0.000184s	0.001244s	0.147910
6	0.000149s	0.001440s	0.103472
7	0.001770s	0.006527s	0.271181
8	0.005013s	0.004372s	1.146614
9	0.015162s	0.008640s	1.754861
10	0.025156s	0.006231s	4.037449
11	0.111354s	0.041052s	2.712511
12	0.203329s	0.076759s	2.648903

可以看到，case8之前由于数据量太小，并行的线程之间通信overhead较大，所以并行速度远小于串行。但从case9开始，基本加速比都能在2以上，取得了一定的加速。下面测试不同线程数目对parallel性能的影响，以case12为例，每次同样运行三次取平均值：

线程数目	运行时间	加速比
1	0.371487s	0.547339
2	0.212068s	0.958790
5	0.128517s	1.582113
10	0.101141s	2.010347
20	0.076759s	2.648903
30	0.086157s	2.359977
100	0.090384s	2.249618

可以看到，最开始加速比小于1，说明并行的占比不大，反而是并行的开销更大；在10～30个线程数目之间，运行速度达到最快；然后略微递减，但仍然保持在2以上。

可以改进的地方

这种并行策略还存在着不少可以改进的地方，比如：

- 加速的效率较低。可以从上表的数据发现，加速比是远小于线程数目的。这可能和树的结构有关，如果树本身并不存在很多可以合并为task的部分，算法很大程度上就会受到串行部分的限制。或者可以说，并行的效率很大程度上会被树的结构限制住。
- 运行时间不稳定。每次算法的运行时间差异比较大，多的时候甚至有几倍的差距。对于最后几个case，加速比一般在2到4之间波动。
- 没有很好的调度策略。这里采用的是dynamic，会比默认策略性能上略有提升，比如对于case12，dynamic策略大概会提速0.01到0.02秒左右，但提升也不是很大。