lab2

PB21051012 刘祥辉

算法设计

1.串行代码

```
Bellman-Ford(Graph)
    输入：一个有向图Graph，其表示为一个邻接矩阵，其中Graph[i][j]表示顶点i到顶点j的边权重。如果i和j之间没有直接边，则Graph[i][j]为∞。
    输出：一个矩阵dist，其中dist[i][j]表示顶点i到顶点j的最短路径权重。

    1. 初始化：
       设n为图中顶点的数量。
       创建一个矩阵dist，大小为n x n。
       对于每对顶点(i, j)，将dist[i][j]初始化为Graph[i][j]。

    2. 主算法：
       对于每个顶点k (从0到n-1)，执行以下步骤：
           对于每个顶点i (从0到n-1)，执行以下步骤：
               对于每个顶点j (从0到n-1)，执行以下步骤：
                   if dist[i][j] > dist[i][k] + dist[k][j] then
                       dist[i][j] = dist[i][k] + dist[k][j]

    3. 返回dist矩阵。
```

1.1 问题分析

可并行化的部分：

Bellman-Ford算法的核心部分是三重循环：

```
plaintext复制代码for k from 0 to n-1
    for i from 0 to n-1
        for j from 0 to n-1
            if dist[i][j] > dist[i][k] + dist[k][j] then
                dist[i][j] = dist[i][k] + dist[k][j]
```

其中内层的两个循环（针对`i`和`j`）可以并行化。对于给定的`k`，每个`i`和`j`的更新操作是独立的，因此这些更新可以并行执行。

不可并行化的部分：

最外层的循环（针对`k`）不能并行化，因为每次迭代的结果都依赖于之前所有迭代的结果。也就是说，必须先完成`k=0`时的所有更新，才能进行`k=1`的更新，以此类推。

可能产生空等的地方：

不均衡的工作负载分配：如果某些线程/进程完成任务较快，而其他线程/进程仍在处理，前者就会处于空等状态。

访问共享资源时：如果并行化时使用锁或其他同步机制，线程/进程可能会因为争夺资源而等待。

负载均衡问题：

负载可以均衡划分。由于每次迭代中，每对`i`和`j`的更新操作是独立的，可以将这些操作均匀分配给不同的线程/进程。例如，假设有`p`个处理器，可以将更新操作分成`p`个块，每个处理器负责其中一个块。这样可以在一定程度上实现负载均衡。

额外的并行化开销：

并行化会引入一些额外的开销，主要包括：

- 线程/进程创建和管理的开销。
- 同步和通信开销：在某些情况下，需要线程/进程之间进行通信或同步，可能会引入额外的时间开销。
- 内存开销：为了减少线程/进程间的竞争，可能需要额外的内存来存储临时结果。

1.2 算法描述

PCA框架

- # 1. 分解（Partitioning）

  分解阶段主要是将计算任务分解成多个独立的子任务。在Bellman-Ford算法中，最适合并行化的是松弛操作。对于每条边的松弛操作可以独立进行，因此我们可以将这些操作分解为独立的任务。

```
c复制代码for (i = 1; i <= m; i++) {
    // 松弛操作
    if (dis[v[i]] > dis[u[i]] + w[i]) {
        dis[v[i]] = dis[u[i]] + w[i];
    }
    if (dis[u[i]] > dis[v[i]] + w[i]) {
        dis[u[i]] = dis[v[i]] + w[i];
    }
}
```

2. 通信（Communication）

通信阶段是指任务之间需要进行数据交换的情况。在Bellman-Ford算法中，每次松弛操作之后，所有线程需要同步以确保所有顶点的最短路径估计值是最新的。这种同步需要在每次迭代结束时进行。

在OpenMP中，可以使用并行for循环和reduction来管理通信和同步。

```
c复制代码check = 0;
#pragma omp parallel for schedule(dynamic, 5) reduction(|:check)
for (i = 1; i <= m; i++) {
    int tmp = dis[u[i]] + w[i];
    if (dis[v[i]] > tmp) {
        dis[v[i]] = tmp;
        check = 1;
    }
    tmp = dis[v[i]] + w[i];
    if (dis[u[i]] > tmp) {
        dis[u[i]] = tmp;
        check = 1;
    }
}
```

3. 聚合（Aggregation）

聚合阶段是将多个任务组合成更大的任务，以减少通信开销并提高计算效率。在Bellman-Ford算法中，可以将每次迭代中的所有松弛操作看作一个大任务，然后使用并行for循环执行这些任务。

这一步已经在之前的分解和通信部分自然地聚合了任务。

4. 映射（Mapping）

映射阶段是将任务分配给实际的处理器。在OpenMP中，这一步通过并行for循环和调度策略实现。通过合理的调度策略（如dynamic调度），可以均衡负载并提高效率。

```
c复制代码#pragma omp parallel for schedule(dynamic, 5) reduction(|:check)
for (i = 1; i <= m; i++) {
    int tmp = dis[u[i]] + w[i];
    if (dis[v[i]] > tmp) {
        dis[v[i]] = tmp;
        check = 1;
    }
    tmp = dis[v[i]] + w[i];
    if (dis[u[i]] > tmp) {
        dis[u[i]] = tmp;
        check = 1;
    }
}
```

伪代码

```
Initialize dis array
for i from 0 to n-1:
    dis[i] = INF
dis[source] = 0

Bellman-Ford algorithm core
for k from 1 to n:
    Backup dis to bak
    parallel for i from 0 to n-1:
        bak[i] = dis[i]

    Relax edges
    parallel for i from 1 to m:
        if dis[v[i]] > dis[u[i]] + w[i]:
            dis[v[i]] = dis[u[i]] + w[i]
        if dis[u[i]] > dis[v[i]] + w[i]:
            dis[u[i]] = dis[v[i]] + w[i]

    Check for updates
    check = 0
    for i from 0 to n-1:
        if dis[i] != bak[i]:
            check = 1
    if check == 0:
        break
```

2.实验结果

强可扩放性分析，即在固定问题规模的情况下（OJ 平台），分析程序在不同核数下的性能表现。分析未能达到线性加速的可能因素。如果出现超线性加速，分析可能原因。

核数	时间/ms
1	48543ms
2	34975ms
4	28457ms
6	21654ms
8	18513ms

分析未能达到线性加速的可能因素

1. **通信开销和同步开销**:
    - 多核并行计算时，处理器之间需要进行数据同步和通信。这些开销在核数增加时会逐渐显现，并且不能忽视。
    - 在多核环境下，尤其是共享内存架构中，缓存一致性协议的开销也会影响性能。
2. **负载不均衡**:
    - 工作负载在不同处理器之间可能分配不均，导致某些处理器的任务较多，而其他处理器较少，从而影响整体性能。
3. **并行化效率**:
    - 并行部分可能没有充分利用所有可用核数，或某些部分代码无法并行化（Amdahl's Law），导致整体加速比受限。
4. **内存带宽限制**:
    - 多核同时访问内存时，可能会受到内存带宽的限制，导致性能无法线性提升。

3.总结

选好合适的算法进行并行化，不要只是并行已有的串行代码