并行计算实验报告

牛庆源 PB21111733

P1 矩阵LU分解

1. 算法设计
   1. 问题分析
      - 可以并行化的部分：
        除法步骤：对每一列元素进行缩放，除了对角线元素。
        消去步骤：消去每一行下面的元素。
        因为这些部分每个循环迭代之间是相互独立的。
        主循环不能并行化，因为每一次迭代依赖前一个结果。
      - 可能产生空等的地方：
        主循环中，每次除法步骤和消去步骤之间，每个线程需要等待其他线程完成当前`k`值的操作后，才能继续下一个`k`值的操作。
      - 负载均衡划分比较好：
        除法步骤中每一个`i`均匀分布。
        消去步骤使用`collapse(2)`将两个嵌套循环合并为一个大循环，负载会均匀分布在多个线程之间。
        提取L和U矩阵的操作也均匀分布。
      - 并行化额外开销：
        线程创建和销毁会有开销。
        `#pragma omp for`之后会有隐式屏障同步开销。
        线程调度和管理的开销。
        缓存一致性和内存带宽竞争会有开销。
   2. 算法描述
      符合PCAM设计方法学：
      1. 划分：
         - 除法步骤中，对矩阵的每一列进行独立的缩放操作。
         - 消去步骤中，对矩阵的每一行进行独立的消去操作。
         - 提取L和U矩阵时，独立提取矩阵中的元素到L和U矩阵。
      2. 通信：
         每次主循环迭代结束时，各个线程需要同步以保证当前列的除法和消去操作已经完成。
      3. 聚合：
         使用了`collapse(2)`聚合嵌套的循环，减少通信开销和同步的频率。

4. 映射：
   OpenMP编程，自动处理了任务映射，实现负载均衡。

具体算法过程：

1. LU分解过程：
   - 对于每一个主对角线元素 `k` 从0到n-1：
     1. 除法：并行地将第 `k` 列的主对角线以下的元素进行缩放。
     2. 消去：并行地将第 `k` 行以下的所有行进行操作，消去第 `k` 列的元素。
2. 提取L和U矩阵：
   并行地从矩阵A中提取L和U矩阵。

2. 实验结果

线程数	总耗时
1	13551ms
2	14109ms
4	12547ms
6	11716ms
8	11213ms

没有达到线性加速，但是随着处理器数的增加，运行时间在减少。

1. 有部分串行操作比如输出输出。
2. 负载问题：LU分解随着迭代进行，参与计算的矩阵区域减小，后期并行任务变少，处理器会闲置。
3. 通信和同步的开销：之前有所提及。
4. 缓存和内存带宽限制导致了线程数从6到8时性能增加不明显。

3. 结论
这次并行化矩阵LU分解使用了OpenMP编程，在编程过程中思考了可以并行化的部分，并尽量一次优化好并行效率。
了解了并行化计算的优点，以及随着线程数增加所带来的性能瓶颈。

P2 单源最小路径问题

1. 算法设计
   1. 问题分析
      - 可以并行化的部分：
        每条边的松弛操作可以并行化，因为每次迭代对边的处理是独立的。

- 可能产生空等的部分：
  由于使用 `#pragma omp critical` 维护 `dist` 数组更新，每次只允许一个线程进入临界区执行对数组的更新操作，其他线程需要等待。
- 不均衡，因为每个节点的更新次数取决于边的数量以及算法的迭代轮数。不同节点完成更新的迭代轮数不同，所以一些线程可能在后期处于闲置状态。
- 会有额外开销：
  临界区的保护会引入额外的同步开销。
  线程创建和销毁，数据的同步也会有开销。

2. 算法描述
   不符合PCAM设计方法学，因为Bellman-Ford算法依赖于顺序执行的动态规划方法，每一个节点的更新依赖于上一轮迭代的结果。
   具体算法过程：
   1. 初始化 `dist` 数组。
   2. 并行化迭代：并行化松弛每条边，如果从源节点经过节点u到节点v的路径比当前已知的路径短，则更新节点v的最短距离。
   3. 输出。

2. 实验结果

线程数	总耗时
1	24700ms
2	27553ms
4	≥35561ms（部分超时）
6	≥35673ms（部分超时）
8	略

并行化开销太大，在线程数增加后耗时增加。

3. 结论
   并行化了Bellman-Ford算法，但是没有办法使用PCAM设计方法学。
   并行化开销严重，使得线程数增加并没有减少耗时，而是增加了耗时，了解了并行化开销大的情况导致的性能下降。

P3 K-means聚类

1. 算法设计
   1. 问题分析
      - 哪些部分可以并行化，哪些不能?
        - 可以并行化：

1. 数据的广播、分发和汇总：使用 `MPI_Bcast` 进行广播操作，在所有进程间同时进行，保证每个进程都有相同的初始数据。使用 `MPI_Scatterv` 和 `MPI_Gatherv` 分发和汇总，每个进程只处理分配给自己的数据部分。
2. 迭代计算中：计算数据点到聚类中心的距离和分配可以在各个进程中并行执行。
3. 局部聚类中心的更新可以被每个进程独立计算。
4. 全局聚类的更新使用 `MPI_Reduce` 并行化。

- 不能并行化：
  1. 中心均值计算：进程0在每个迭代结束时计算新聚类中心的均值。
  2. 全局距离计算。

- 可能产生空等的地方：
  全局中心的更新：每次迭代结束后，所有进程需要等待进程0更新完聚类中心并广播。
- 负载均衡问题：
  1. 数据分配可能导致负载不均衡，尤其是当数据数量不是进程数量的整数倍时。
  2. 中心更新：有些中心的数据点可能更多，导致更新操作负载不均衡。
- 并行化额外开销：
  1. 通信开销：广播和汇总操作会带来通信开销。
  2. 同步开销： `MPI_Bcast` 和 `MPI_Reduce` 会带来同步开销。

2. 算法描述
   大体符合PCAM设计方法学：
   1. 划分：
      - 原始数据被划分为多个局部数据块，每个进程处理自己的数据块。
      - 计算任务（距离计算和聚类分配）也被划分，每个进程独立计算其局部数据块的分配结果。
   2. 通信：
      - 广播全局信息（数据维度、聚类中心数量和初始聚类中心）。
      - 使用 `MPI_Scatterv` 分发数据。
      - 每次迭代中，用 `MPI_Gatherv` 汇集分配结果，用 `MPI_Reduce` 聚合局部新聚类中心和计数。
   3. 聚合：
      - 使用高效的MPI操作减少通信和同步开销。
      - 在进程0计算新的聚类中心均值，并广播给所有进程。
   4. 映射：
      - 每个MPI进程处理分配给它的数据块，计算局部结果并参与全局通信操作。

   具体算法过程：

1. 初始化MPI。
2. 数据的读取和广播：在进程0中读取数据集大小、维度和聚类中心的数量，然后广播这些信息给所有进程。
3. 数据分发：将数据集划分成多个块，每个进程处理一个数据块。通过 `MPI_Scatterv` 实现。进程0根据总数据量和进程数量计算每个进程的分配，并将数据分发给各个进程。
4. 迭代更新：
    - 每个进程计算其数据块中每个数据点到所有聚类中心的距离，确定最近的中心。
    - 使用 `MPI_Gatherv` 将所有进程的分配结果汇总到进程0。
    - 每个进程独立计算其数据块中新聚类中心的部分贡献。
    - 使用 `MPI_Reduce` 汇总所有进程的新聚类中心贡献，并在进程0计算新中心均值。
    - 使用 `MPI_Bcast` 将新的聚类中心广播给所有进程。
5. 计算总距离：在迭代完成后，进程0计算所有数据点到其聚类中心的总距离。
6. 结束MPI环境。

2. 实验结果

在代码内部使用：

```
// 设定参与计算的进程数
int active_procs = std::min(size, 4); // 设定最多4个进程参与计算
```

来规定使用的进程数，同时在代码的其他部分也做相应修改：

核的最大数量	总耗时
1	Runtime Error
6	Runtime Error
8	6936ms
10	7503ms
12	7819ms

代码修改为单核时会出现 `Runtime Error` 报错，一直增加核数量直到8报错消失。

没有线性加速也没有超加速。并行开销大。

3. 结论

使用了MPI并行编程，学习了MPI的函数以及一些核的调配和广播。同时加深了自己对K-means算法的理解（可否并行化）。

P4 稀疏矩阵乘法

1. 算法设计
    1. 问题分析
        - 哪些部分可以并行化，哪些不能?
            - 可以并行化的部分:
                1. 稠密矩阵和稀疏矩阵的输入：输入稠密矩阵和稀疏矩阵非零元素的部分可以并行化读取。
                2. CUDA kernel 函数 `sparseDenseMatMult` ：每个线程处理结果矩阵的一个元素。当前线程通过块和线程索引确定需要处理的矩阵位置。
                3. `sortSparseMatrix` 函数可以并行化实现排序，但是这里是串行实现的。
            - 不能并行化的部分:
                1. 输入的处理和排序函数:
                    - 输入处理（包括排序）和CUDA内存分配、数据传输等部分基本上是串行执行的。
                2. CUDA内存管理:
                    - `cudaMalloc` 和 `cudaMemcpy` 操作是串行执行的，尽管多个操作可以并行提交，但实际执行是串行的。
        - 可能产生空等的地方:
            1. CUDA kernel执行：由于稀疏矩阵的稀疏性，每个线程可能不会执行相同数量的乘法累加操作。一些线程可能只需要处理很少的非零元素，而其他线程可能需要处理更多的非零元素。这样会导致线程之间负载不均衡，从而产生空等。
            2. 内存拷贝：从主机到设备以及从设备到主机的数据传输会有延迟。这些操作在调用 `cudaMemcpy` 函数时，主机会等待数据传输完成。
        - 负载没有很好的均匀划分:
          稀疏矩阵中非零元素分布不均匀，导致不同线程处理的乘法累加操作数量不同，负载不均衡。
        - 并行化额外开销:
          CUDA内存管理开销：CUDA内存分配、内存拷贝以及CUDA kernel启动会有一定的开销。
    2. 算法描述
       符合PCAM设计方法学:
        1. 分解:
            - 使用Kernel函数`sparseDenseMatMult`，结果矩阵中的每个元素都是独立计算任务，由各个线程独立完成。
        2. 通信: （ `cudaMemcpy` ）
            - 将输入的稠密和稀疏矩阵数据从主机传输到GPU。

- 将结果矩阵数据传回主机。

3. 聚合：
   - 使用块级别的并行，而不是每个线程处理一个结果元素，可以提高负载均衡。
   - 使用CSC格式压缩。
4. 映射：
   - 通过定义线程块和网格大小（`dim3 threadsPerBlock(16, 16);`），将每个计算任务映射到具体的CUDA线程和块上。

2. 实验结果

修改线程块中的线程数：

线程块尺寸	总耗时
1×1	⩾46305ms（部分超时）
2×2	11298ms
4×4	5218ms
8×8	5799ms
16×16	4970ms
32×32	5635ms

在线程块尺寸从 1×1 扩大到 2×2 时，出现了超线性加速现象，可能的原因为：

1. CUDA核心在 1×1 时，每个线程块只有一个线程，导致GPU的并行处理能力未得到充分利用。而 2×2 时，更多的GPU核心被激活和利用，显著提升了计算效率。
2. 内存访问模式：当线程块大小增加时，更多的线程可以协同工作，利用共享内存和全局内存进行合并访问，可以显著提高内存带宽利用率，减少了内存访问延迟，提高整体性能。
3. 指令级并行：线程块大小从 1×1 到 2×2 时，能更高利用指令并行性，使多个指令在不同的线程中重叠执行，可以减少每个线程的等待时间，提高整体计算效率。

之后在 16×16 时运行效率最高，并在 32×32 时耗时增加，表示这时并行开销大于并行计算提高的效率。

3. 结论

CUDA使用GPU加速，通过分配线程块大小来控制线程数。学习了CUDA有关的函数以及分配。
使用CSC压缩稀疏矩阵，提高运算效率。
没有并行化排序函数，可以并行以提高效率。