矩阵 LU 分解

算法设计

问题分析

串行算法：

```
LU(Mat A):
    for k = 1 to n:                                  //外层循环不能并行，循环内存在数据依赖
        for i = k + 1 to n:                          //可以并行
            A[i][k] = A[i][k]/A[k][k]
        for i = k + 1 to n:                          //可以并行，同时可以与内层循环合并为一个循环
            for j = k + 1 to n:
                A[i][j] = A[i][j] - A[i][k] * A[k][j]
    Mat L, U
    for i = 1 to n:                                  //可以并行，同时可以与内层循环合并为一个循环，但是需要将最后的L[i][i] = 1放在循环外单独并行
        for j = 1 to n:
            if i > j:
                L[i][j] = A[i][j]
            else:
                U[i][j] = A[i][j]
        L[i][i] = 1
```

- 在omp循环并行化结束同步时可能会产生空等。
- 基本能够做到负载均衡。因为循环内都是固定的赋值运算语句，每次循环时间开销相近，在数据量n大于处理器数时就能够做到均衡划分。
- 额外开销：主要体现在线程的创建、通信、同步、负载分配。

算法描述

并行算法：

```
LU(Mat A):
    for k = 1 to n:
        for i = k + 1 to n par-do:
            A[i][k] = A[i][k]/A[k][k]
        for i = k + 1 to n par-do:
            for j = k + 1 to n:
                A[i][j] = A[i][j] - A[i][k] * A[k][j]
    Mat L, U
    for i = 1 to n par-do:
        for j = 1 to n:
            if i > j:
                L[i][j] = A[i][j]
            else:
                U[i][j] = A[i][j]
    for i = 1 to n par-do:
        L[i][i] = 1
```

- PCAM算法拆分：本问题中可以将最内层循环中的语句看作时一个小任务，在第一个循环中存在数据依赖，只能在内层循环中并行；后两个循环中不存在数据依赖，可以直接并行。
    - 通信：openmp通过共享内存交换数据。
    - 组合：可以通过openmp设置schedule的chunksize实现讲小任务合并为较大的任务。
    - 映射：主要通过openmp的schedule设置任务调度方式。

实验结果

通过提供的测试数据循环运行10次LU分解函数得到时间花费。

线程数	1	2	4	8
时间(ms)	27973	14419	9044	6390
加速比	1	1.94	3.09	4.37
利用率	1	0.97	0.77	0.55

可以看到随着线程数的增加，时间花费逐渐减少，加速比逐渐增大，但是利用率逐渐减少，到线程数为8时只剩下接近一半。可见程序并未能到达线性加速。

原因主要是随着线程数的增多，每个线程并行的数据量减少，而总的线程的创建、通信、同步、负载分配等开销增加，由amdahl定律知，固定负载下，加速比会趋于一个上界，体现在测试时间上则是加速比增加逐渐减缓，利用率不断下降，无法到达线性加速。

结论

在需要同时考虑时间开销与计算机资源的利用时，应该设置一个合理的线程数，过大，则资源利用率迅速下降，加大成本；过小则会导致时间开销较高

单源最小路径问题

算法设计

问题分析

串行delta-stepping算法：

```
for v in V:
    tent(v) = ∞
relax(s, 0)
//算法主体
while isNotEmpty(B):
    i = min{j ⩾ 0: B[j]≠∅}
    R = ∅
    while B[i] ≠ ∅:
        Req = findRequests(B[i], light)          //可并行
        R = R ∪ B[i]
        B[i] = ∅
        relaxRequests(Req)                       //可并行
    Req = findRequests(R, heavy)                 //可并行
    relaxRequests(Req)                           //可并行
```

```
15
16
17
18  def relaxRequests(Req):
19      for (w, x) in Req:
20          relax(w, x)
21
22  def findRequests(V', kind in {light, heavy}):
23      return { (w, tent(v) + c(v, w)): v in V' and (v, w)满足kind条件 }
24
25  def relax(w, x):
26      if x < tent(w):
27          B[ ⌊ tent(w)/Δ ⌋ ] -= {w}
28          B[ ⌊ x/Δ ⌋ ] += {w}
29          tent(w) = x
```

- 并行分析：可并行主要存在于以下三个部分：
    - 查找light和heavy边时需要遍历所有的边，分别存入light和heavy以备使用。
    - findRequests：对于从B[i]引出的的每条light边，都需要计算由此得到的目的节点的新tent也即距离值。
    - relaxRequests：对于Req中每个元素都需要对其进行relax操作
- 空等、负载、并行开销与实验1基本相同

算法描述

并行算法：只需并行化上述串行算法的可并行部分即可

- PCAM算法学：同为openmp，过程与实验1基本相同

实验结果

通过提供的测试数据循环运行10次delta-stepping算法函数得到时间花费。

线程数	1	2	4	8
时间(s)	6.637	5.25	4.519	4.274
加速比	1	1.26	1.47	1.55
利用率	1	0.63	0.44	0.194

可以看到随着线程数的增加，时间开销和加速比的指标提升的非常有限，而利用率迅速下降，说明算法并行化的效果并不好，到线程数为8时只剩下不到20的利用率。可见程序并未到达线性加速。

原因主要分为以下几点：

- 随着线程数的增加，总的线程的创建、通信、同步、负载分配等开销增加。尤其是算法一次迭代中存在多次并行需要多次同步，极大的增加了开销。
- 算法只有循环内的一部分可以并行化，外层循环无法直接并行，循环内还有相当数量的串行部分。这些因素表明算法负载中的可并行负载占比不大，由amdahl定律可知加速比的上界较小，随着线程数的增加很容易接近上界，从而加速比增长缓慢、利用率迅速下降，无法到达线性加速。

- 可并行化的循环中每次迭代开销不确定，可能相差很大。以relax为例，需要查找B[i]中的指定数据并删除之，那么当B[i]中数据量较大与数据量较小时的开销可能相差巨大从而导致线程同步时出现较长的空等。

此外，本算法的性能与delta的选取具有强相关性，而delta的选取很大程度上需要取决于数据的分布情况，因此根据输入数据动态更新delta会是一个可能的改进方向。

结论

直接根据已有的串行算法设计并行算法，并行算法的性能很大程度上依赖于串行算法。对比实验1，2；实验1串行算法中大部分都可并行化，因此并行算法性能较串行算法有很大提升；而实验2中相当部分不可并行，导致并行算法性能不佳。

K-means 聚类

算法设计

问题分析

串行算法

```
K_means():
    read N*M-dimension datas
    randomly generate K points as centers
    while iteration < max:                                  //不可并行
        for data in datas:                                  //可并行
            for center in cluster centers:
                calculate distance of data and center
            determine center that data belongs to
        for center in cluster centers:
            calculate new coordinate of center

```

- 并行分析：显然迭代之间不能并行，每次迭代内可以通过将数据均与分发给各个进程实现并行。
- 空等：由于各个进程的负载时提前确定的，不能像openmp动态分配，因此不能保证负载完全相同，在同步时必然会出现一定程度的空等；此外，根进程处理串行分量依据分发数据时其余进程也必然会出现空等。
- 负载：虽然负载提前确定，不能保证完全相同，但是由于本算法中每个数据迭代一次的开销变化不大，因此通过均匀分发数据的方式能保证一定程度的负载均衡。
- 并行开销：分为消息传递、进程同步和进程管理等，其中又以消息传递为主。

算法描述

1. 主进程读取数据，均分并分发数据，生成随机中心点并广播。
2. 各个进程根据自己分配的得到的数据和主进程广播的中心点，计算数据点与各中心点的距离、确定数据点所属中心点、计算各中心点包含的数据点的坐标之和、将数据汇集至主进程。
3. 主进程收集各进程的数据并据此计算新的中心点坐标，随后再次广播。若迭代次数大于100，退出；否则执行2。

- PCAM算法学

- 划分：在本实验中可以将每次迭代中对每一个数据点的所有操作看作一个小任务，且每个小任务的任务量近似相同。
- 通信：由主进程负责初始数据的加载、数据的散播与收集。
- 组合：通过给各个进程分配尽量均匀的数据量组成每个进程的负载。
- 映射：每个进程负责一组数据的处理。

实验结果

通过提供的测试数据循环运行10次K-means算法得到时间花费（此处是重复运行程序，包括输入输出，而非只运行函数）。

进程数	2	4	8
时间(s)	1.5357	1.7125	2.2438
加速比（此处以进程数2为基准）	1	0.90	0.68
利用率（此处以进程数2为基准）	1	0.45	0.17

可以看到随着线程数的增加，时间开销反而增加，加速比与利用率已失去讨论意义，也显然不是线性加速。主要原因是数据量太小，只有200×7，并行的收益太小，又由于每次迭代中有多次数据的广播、收集和同步，这使得并行带来的通信和同步开销超过了并行的收益，从而时间开销反而增加。

结论

算法的并行化并不能总是带来收益，尤其是当数据量较小时，并行化带来的开销可能会超过其带来的收益，因此不能盲目的不分实际情况的运用并行，运用并行算法时需要着重考虑数据规模。

稀疏矩阵乘法

算法设计

问题分析

- 并行分析：本实验中矩阵乘法需要计算M行×P列，因此可以用$M * P$个线程并行执行，每个线程单独负责一个行向量与列向量的内积（实际线程数可能会小于$M * P$，此时一个线程负责多个行列内积）。
- 空等：由于有一个矩阵时稀疏矩阵，线程的实际负载取决于列向量中非零元素的数量。因此，不同数量的非零元素很可能会导致空等。
- 负载：由上可知并不能很好的保证负载均衡，可能矩阵中的非零元素都集中于部分列向量中导致负责这些列向量的线程负责远大于其他列向量。
- 并行开销：主要包括内存在CPU和GPU之间的复制

算法描述

1. 将coo格式的稀疏矩阵转换为csr格式，后者能够更方便的通过稀疏矩阵元素找到稠密矩阵中对应的相乘元素
2. 并行地求稠密矩阵的每一行和稀疏矩阵的每一列的内积并赋值给结果矩阵。

- PCAM算法学

- 划分：理想情况下每个线程负责一个结果矩阵元素的计算，实际上线程数小于$M*P$，此时一个线程计算多个结果。
- 通信：主要通过全局变量和共享变量等进行通信。
- 组合和映射：每个线程实际负责多个结果的计算。

前已提及，稀疏矩阵不同列向量中非零元素的数量很可能会导致负载不均进而影响算法性能，一种可能的改进方向是依据csr存储的矩阵信息确定非零元素较多的列，将其拆分为多个子任务。此时不同的线程写入同一个数据需要原子操作以及更复杂的复杂分配方案，这会带来额外开销。

此外，考虑较极端的情况，M和P很小而N很大，如$(2, 1000000) \times (1000000, 2)$，若仍采用上述算法实际只会有四个线程并行，效率很低；即使是采用前面提到的改进方案，对结果矩阵的原子写操作也会是串行的，并不会改善算法性能。此时考虑拆分列向量，每个线程负责一部分，最后将结果归并。同样，此方案也会带来归并的额外开销，在本次oj中效果不佳。

实验结果

在本次实验中最佳时间约为3.5s。一般情况下，线程数会小于$M \times P$，且本实验中并行的额外开销相对固定（内存复制），因此可以预测，当线程数在一定范围内增加时，算法会取得较好的性能提升。但不会到达线性加速，因为存在额外开销以及可能的复杂不均，后者在线程数增加时将越发明显。

结论

负载均衡时并行算法设计中一个十分重要且较为棘手的部分，当负载不均时，承担了最大负载的线程将成为整个算法的性能瓶颈。