使用串行，OpenMP，pthreads, MPI用k-means算法处理图像

单暖乔 21307130395

我使用了串行，OpenMP，pthreads，MPI四种方法实现了k-means算法处理图像。因为我之前接触到的时候觉得这个处理之后的图像很有趣。

1.文件说明和环境配置

我是在windows系统的vscode里边做的。这个作业的代码逻辑在两个文件夹中，分别叫k_means和k_means_accelerate。其中k_means是放k-means的正常实现的代码的，k_means_accelerate中放的是测试性能用的代码的。每个源文件都可以在vscode上方菜单栏中点击terminal直接Run build task，之后使用下方终端运行。大部分的就直接.\{文件名}就可以，但是mpi要在下方的terminal中输入mpiexec -n {进程数} {文件名}。这里的.vscode环境设置也在压缩包中一并打包上传了。

2.代码思路

串行

这里就是正常的k-means算法，也就是初始化质心之后，计算每个数据点到这些质心的欧式距离，之后找到距离最小的质心作为数据点的分组。把所有的数据点分好之后，对每个分组的中心点，算出这些分组里的数据的平均位置作为中心点的新位置，之后再重复上述的分组和更新中心点位置的步骤，直到中心点的位置的移动变得小于一个阈值。只是实际操作之中，为了能够尽快算，有的时候会设置迭代的上限。

OpenMP

在串行的基础之上，加入了一些并行的逻辑。这里主要在读入数据，计算数据点的分组，更新质心的位置以及输出的这些循环中用的并行。只是在更新质心位置的这个循环之中，newcost需要把所有线程的cost进行累积，因此设置为规约变量。

pthreads

这个是在串行的基础之上把像素点进行分组这里变成了并行的，是按照块的形式划分（虽然在一维上可能形容不贴切，但是意思就是说把数据裁成一段一段的，一个线程分到一段数据）之后每个线程要写的数据是没有交集的，因为像素点分配是没有交集的。由于同步问题略困难，因此初始版本比较保守，在循环中每次进行分组计算之前把线程创建出来，之后同步的时候再把线程结束掉。由于看起来效果不是很好，我就又把更新质心这一步加了并行，但是这一步可能会因为应用中划分的组数比较少（少于创建的线程个数这种而失去作用，因此这个方法并不普适，但是对于这里8颜色分组的，是可以这样做的）。之后我还是决定应对这个同步的问题，我有做了一个拒绝频繁创建结束的版本，这个需要使用pthread_barrier来实现，注意同步障的使用位置，这里需要在分组之后设置同步障，要在更新质心之前确定每个数据点的分组情况已经更新过了；在每个do-while循环结束一次之前进行同步，因为要确保质心更新完毕了再进行新的一轮计算；之后还有一个很容易忘掉，就是在循环开头也要加，如果不加，有的线程会比较快速，立刻把质心移动的距离之和oldCost给更新了（oldCost=newCost），而还没有到达while循环判定条件的线程可能就会发现newCost - oldCost> 1e-6 认为可以结束掉这个线程了，就会乱套，因此这里的同步障很需要注意。

MPI

这个也是在串行逻辑的基础之上进行的改动。主要是在计算数据点的分组和计算新的质心的位置这两个步骤进行了并行，也就是对数据点的处理上进行了并行。数据的划分是循环划分的，这样更有利于负载的均衡。注意，在读数据和写数据的这个过程我是没有使用并行逻辑的，因为要不然通信也会是个开销，这里用了一些部分的冗余计算和串行计算代替了。这个代码的整体思路是很易懂的，但是在这个实现起来需要很注重细节。比如在最后进行图像输出的时候由于每进程分别保存了一些数据的分组信息，但是要由一个进程输出的时候要使用通信或者是冗余计算，这里我用的是冗余计算的方法实现的，还有就是一些数据（比如rgb颜色和属于一个分组的数据点的个数）的加和汇总之类的，要注意不要漏项。同时，对于输入数据的读取之类的地方，如果使用了多线程的方式也要注意汇总，虽然我自己用的是冗余计算。

3.代码说明

我每个版本的逻辑是比较相近的

Pixel是像素点的结构体

distance是用来算欧式距离的

initCentroids是用来初始化质心的

parallelKMeans是用来进行并行的k-means的，这个代码负责计算出质心也就是中心点的位置的，有些版本的（除了MPI)的会带有计算数据的分组的功能，而MPI的版本由于是数据通信模型，这里会在main函数中实现数据的最终分组

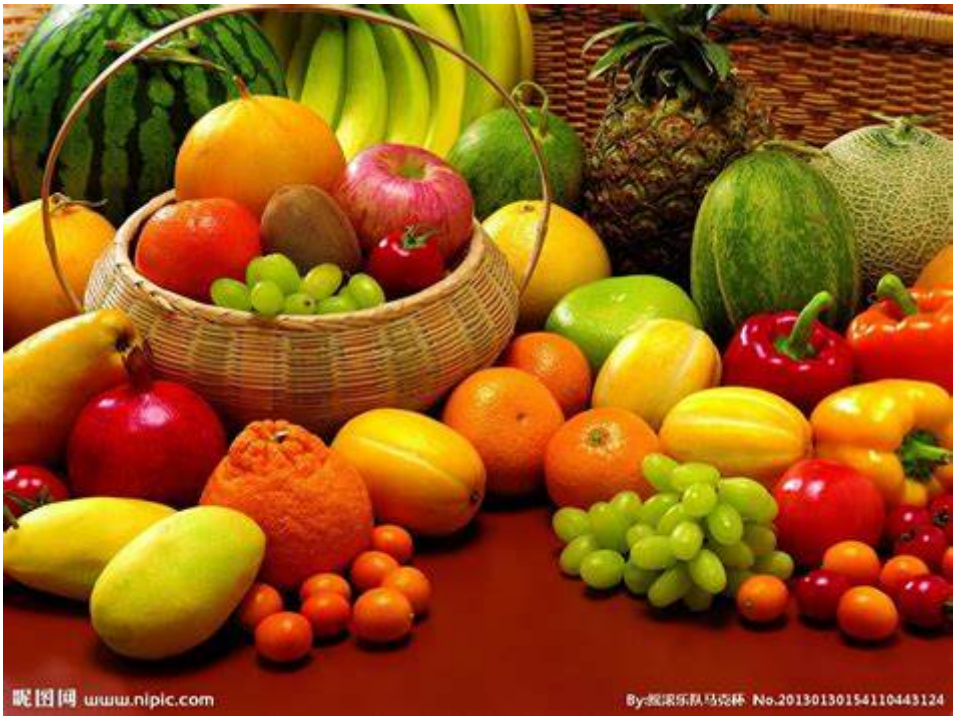
4.实验结果

我设置分组的个数为8，这些数据是我强制迭代500次运算得到的结果，分别对串行，并行中的2个线程/进程，4个线程/进程，8个线程/进程的情况下的单个指定图片的运行结果。

运行时间	串行/1个线程	2个线程	4个线程	8个线程	16个线程
串行	31.1133	NaN	NaN	NaN	NaN
openmp	30.9997	16.6583	9.79791	8.07416	6.66538
pthreads	60.8927	35.9932	16.9428	14.823	13.5964
pthreads_iter	66.0036	30.4689	15.646	12.9308	NaN
pthreads_big_iter	32.6032	16.7753	9.32009	8.17098	NaN
MPI	29.9376	16.3308	9.41732	7.89902	6.43951

加速比	2个线程	4个线程	8个线程	16个线程
openmp	1.8677	3.1755	3.8534	4.6679
pthread	0.8644	1.8363	2.0990	2.2883
pthreads_iter	1.0211	1.9886	2.4061	NaN
pthreads_big_iter	1.8547	3.3383	3.8078	NaN
MPI	1.9052	3.3038	3.9389	4.8316

输入



输出

可视化和控制变量

5.结果分析

在上述的实验中可以看到openmp和mpi和pthreads中的big_iter的效果差不多，要好于pthreads前两个版本。虽然之后的pthreads也做了一些改进，也比改进之前的好一些，但是仍然比别的并行算法慢一些。

openmp的内部实现是不透明的。而mpi的实现中，进程创建之后是做完整个的流程之后才结束掉的，pthreads的初始版本虽然是用的线程，创建的开销也小于进程，但是每次的大循环体中，也就是每次进行一次数据分组和质心更新，都要进行一次线程的创建和结束，iter改进之后是进行两次的创建和结束，这个开销是不能忽略的，更何况是用一个循环来依次结束掉这个线程的。而最终版本的big_iter是直接把创建线程的这个任务给到了循环的外边，因此避免了很大的线程管理的额外开销。而之所以iter改进的

pthreads会比原始版本的好，是因为虽然一次外层循环中会多创建一次线程，但是其带来的并行优化要更加的重要。

之后在线程数量越多的时候，计算速度是越快的，但是随着线程数量的增加，性能的提升速度会逐渐的变慢。

这个是因为线程或者进程越多，进行同步或者是通信的开销就也会随之增大，这会影响到并行算法的性能。

6.亮点和工作量

亮点

这里做了可视化。使用了opencv对图像进行处理，看起来更加的直观不枯燥。

这里的测试性能和正常的算法实现用了两套代码，因为在性能测评的时候要控制计算量相等，因此在算法终止条件的判定上，测试性能的代码和真正的算法用的代码会有些许的出入，这个在不同方法实现的代码中改动方式是有差异的，因此索性做了两套完整的代码。

使用了3种算法实现。方便对不同的算法进行横向比较。也看到实现同一种功能的多种实现思路。

对于pthreads做了2步优化，使得性能提升。

工作量

其实这个中很大的一个工作量就是配环境，以及分别配好的环境怎么能同时的好用，opencv和omp和mpi这几个环境融合到一起这个过程其实就比较的费时费力。

之后就是写代码的部分，测试性能和正常的算法两个文件夹中的源代码都分别大概是6个，代表这个问题的不同实现方法，虽然有的源代码会比较相似，但是仍需要时间处理其中的差异，不断对比。

对于一些pthreads和mpi的这种比较需要注意细节的，要更加注意代码的逻辑。

7.总结

一些可以改进的方向

但是k-means也有它的一些局限，因为这个算法的逻辑就是数据点分组和质心位置的更新两个步骤交替进行的，这会天然的带有顺序性，因此，在代码实现的时候就需要显式或者隐式加入同步的机制，这会降低一部分并行的性能，但是总体上还是可以明显的观察到并行处理之后算法性能的提高的。

此外，在pthreads的任务划分上，由于最后一个线程会把余数的部分都算到自己的工作量中，这个会不如我在mpi方法中的循环划分出的工作量均衡，而又因为有各个同步障，这会比较拉低性能。因此这里有优化的空间。

而在mpi中，在最后一次算分组准备进行输出的时候，可以尝试用做好的局部数据进行通信汇总数据，减少一部分冗余计算和串行的部分，这个也是可以进一步优化性能。

总结和感悟

我当时是学习了k-means算法，又看到了这个k-means算法对图像的这个应用之后，认为这个算法功能会比较有趣，并且数据点之间会存在一定的并行性，可以尝试使用并行算法提高处理效率，因此选择做实现这个算法。而这些并行算法中，确实是可以看到并行化算法之后运算速度的提升的。在实现多个方式的算法，逐渐优化算法的过程中，我也学到了很多，体会不同的编程模型之间的思路和实现的不同，对我很有挑战性。