高斯消元法的多线程实现

计算机科学与技术
1511186
梁宸

February 25, 2020

1 简介

本次编程作业使用C++语言实现了pthread库的多线程版本的高斯消元法程序, 实验处理器环境为intel i7, 操作系统环境为Linux Mint 18.3 64bit.

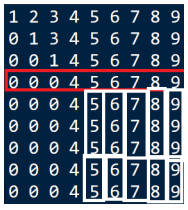
2 算法设计

2.1 划分子任务

朴素的高斯消元法的大概流程是, 对于输入矩阵的每一行, 先将该行的所有元素都除以该行第一个非零元素; 然后对于其下的所有行, 再进行消除. 分析朴素算法各个环节的依赖关系, 第一阶段中对当前行的scale操作一定要先于第二阶段; 而第二阶段中下面各个行的消除之间并没有依赖; 这样则可将子任务分成两类, 分别对应算法的两个阶段. 这样多线程的高斯消元法的大概流程为, 输入矩阵的每一行对应一轮, 每轮划分有两类子任务, 其一是对于该行做scale操作, 其二是对下面的所有行做消除操作. 如果采用列主映射[1], 并按列划分子任务块, 那么所有第一类子任务一定要先于其同一列第二类子任务完成.

2.2 任务分割

考虑到列主映射的特性, 将矩阵划分为宽为1, 等长的块, 每块对应一个第二类子任务,



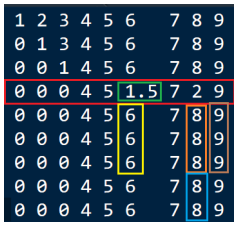
[1]即同一列的元素存在内存的连续区域

如图所示. 其中红色部分为当前行, 白色即为长度为3时的块. 考虑到AVX的YMM寄存器的长度能容纳8个单精度float, 那么块的长度应为8的倍数. 那么对于图中样例, 一类子任务有将当前行的5, 6, 7, 8, 9分别除以4四个; 而每一列的一类子任务结束后, 则可以解锁该列的第二类子任务, 也即是将每块中的元素按高斯消元法的思路消除. 当所有子任务结束后, 开始下一行的新一轮.

2.3 线程任务调度

每当一个线程完成一个子任务后, 他会执行调度函数来寻找下一个子任务. 在考虑工作线程的任务分配时, 考虑到i7处理器的L1, L2缓存为核心独享, 虽与线程级有些差别, 但仍然一个自然的想法是, 每个工作线程尽可能地按列的顺序访问矩阵执行子任务, 以提高缓存命中率.

于是设计在每一轮开始时, 分配所有工作线程依次做所有列的第一类子任务; 每当线程完成一个任务后, 调度算法会按列的顺序, 找到该列之后第一个未完成的块[2], 如果该块所在的列已经被scale, 那么让该线程直接执行该块的工作, 让该线程执行对应的第一类子任务, 如果此任务已经有人在做, 则寻找下一个未完成的块. 如果所有未完成的块都这样, 那么让该线程等待某个第一类任务的完成, 再执行其下的第二类任务.



如图所示, 如某个线程刚刚执行完绿色部分的第一类子任务, 即把6 / 4, 那么他接下来会调度到黄色部分的块对应的第二类子任务, 将它们消除; 如果某个线程完成了橙色部分的第二类任务, 并且蓝色块对应的任务也已被完成, 那么它会找到棕色块, 并且棕色块对应的列首9尚未被scale, 那么该线程会被调度执行该列的第一类任务, 即9 / 4.

考虑到在这样的调度算法中, 查询某个块之后第一个未完成块的操作是相当频繁的, 于是使用并查集的数据结构, 并引入路径压缩优化. 详细的原理在此不赘述. 这样在n个块的并查集中进行一次查询或修改操作的时间复杂度为O(α(n)), 其中α(n)为阿克曼函数的反函数, 它的增长是相当缓慢的, 在一般的问题中, 可以粗略认为时间复杂度为常数级.

2.4 线程同步机制

采用master-worker的模式, 一个master线程负责初始化和协调工作, 多个worker线程做两类子任务. master线程执行的主循环为对于行的枚举, 称作一轮. 每一

[2]未完成即是指该块未被消除过.

轮master会先做一些初始化工作, 比如分配worker线程最初的工作, 块状态的初始化等等, 然后等在barrier1前. barrier1用来确保在worker真正开始工作前, master已经做好了这些初始化工作. barrier1过去后, master立刻等候在barrier2, barrier2用来确保本轮的所有工作已经完成, 并且所有worker已经结束工作. barrier2过后, master即可以做本轮的收尾工作;

worker线程由master线程在主循环执行前创建, 程序结束时销毁, 执行的是工作循环. 工作循环每轮要先等候在barrier1前, 等待master完成初始化工作. 随后worker进入内层循环, 它不断的调度寻找下一个任务并完成. 当它发现没有未完成的块时, 即说明该轮工作自己已经没什么能做的了, 于是退出内层循环; 若它发现有尚未完成的块, 但该列的第一类任务正在由别的线程做, 并且接下来没有能直接做的任务, 那么它不得不等待这个线程, 使用pthread cond机制实现线程的条件等待与唤醒.

在调度中, 需要数据结构来维护本轮各个块的完成状态, 使用pthread mutex来保证各个worker对它的读写是互斥的.

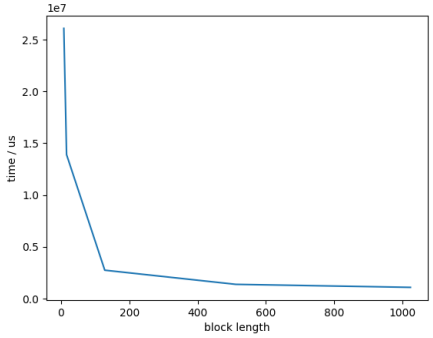
2.5 初步分析

在上述算法中, 块的长度是一个尤为关键的参数. 它决定了每个子任务的粒度, 影响了线程通信的密集度和算法的加速比和可伸展性等等指标. 当块长度很大, 比如超过矩阵长时, 相当于每次分配给每个线程一列, 只有线程做完一整列的操作后才会调度, 产生通信. 这样虽然通信次数会很少, 但当线程数增大时, 能真正工作的线程数受矩阵长度限制的, 一旦线程数大于矩阵长度, 多余的线程并不能获得工作. 此外, 粗粒度的任务划分不利于线程间的负载均衡. 所以这样会减少线程通信, 但也会降低算法的可伸展性; 当块长度很小时, 比如最小为8时, 子任务的划分达到最小的粒度, 线程对于每个块要做的只有几个AVX操作, 算法的并行度大大提高, 线程间的复杂均衡也改善, 但这会大大增加线程间通信次数. 于是这样会提高可伸展性, 但降低算法效率.

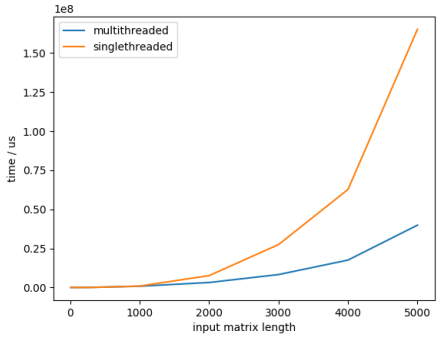
3 结果与分析

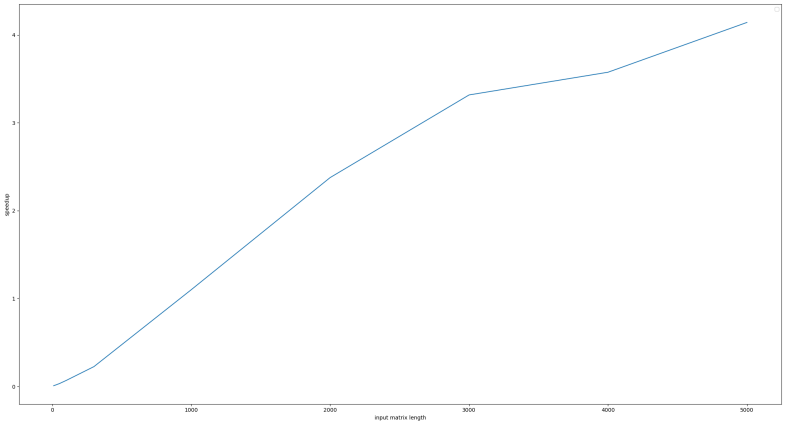
首先考察块长度对算法效率的影响, 在中等规模的输入, 工作线程数为7时[3], 结果如下:

[3]这主要是因为i7处理器为8线程.



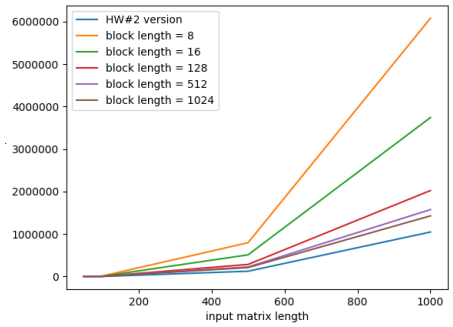
可以看到该算法在块长度增加时, 效率大幅提高.于是暂且先用较大的块长度运行程序, 获得与单线程的AVX高斯消元法程序的对比, 结果如下:





可以看出, 在直到最大输入为5000*5000的矩阵时, 加速比的上升趋势依然没有显著消失, 这说明应该使用更大的输入以测试, 然而这有点超出测试处理器的性能负担范围, 运行时间有点长. 不过理论估计, 测试最大线程数为7, 这与输入规模相比稍小, 若输入规模继续增大, 应能接近线性加速比.

一个差强人意的问题是, 在结果中块长度似乎只和算法效率负相关, 而并没体现出改善负载均衡等预计效果. 怀疑是否为一些初始化等常数因素或者并查集等数据结构操作带来的额外开销, 于是设置输入矩阵长度为1000, 工作线程数为1, 测试在不同的块长度下, 本程序与上次作业中单线程的AVX程序执行时间对比, 结果如下:



可以看到, 随着块长度的上升, 与单线程版本的程序执行时间差也显著上升. 考虑本次程序与上次程序的区别, 除主线程的一些固定开销外, 唯一的一个工作线程在每个块工作结束后都要调用pthread mutex最终进入系统调用. 那么这些系统调用的次数应该为(矩阵大小/ 块长度)次, 这与图中差值的接近线性相关.

4 改进思路

基于之前的结果与分析, 可以发现该算法的最大瓶颈在于工作线程对块状态的互斥访问, 而事实上这些访问不一定是必须互斥的, 比如可以使用读写锁或者更细粒度的锁来达到更少的阻塞. 此外也可以使用自旋锁来避免阻塞, 减少等待时间.