**高性能计算导论HW5**

**计71 张程远 2017011429**

**PA 1**

本题要求优化HW3中完成的代码，使之效率提升。最开始的时候，我采用了OpenMP和MPI结合的实现方式。在3.5中，每个进程计算矩阵时加入了omp处理，即

#pragma omp parallel for num\_threads(8)

for(int local\_j=0;local\_j<n;local\_j++){

for (int local\_i=0;local\_i<local\_n;local\_i++){

/\* calculate \*/

}

}

在3.6中具体的实现如下面所示：

#pragma omp parallel for num\_threads(4)

for(int i=0;i<local\_n;i++){

for(int j=0;j<local\_n;j++){

loc\_ans\_vec[i]+=local\_A[i\*local\_n+j]\*loc\_vec[j];

}

}

但是实际上最后获得的时间相比于优化前并没有提高太多。以3.5题为例，分别测试N=20000，40000时这段代码块所耗时间，结果如下：（线程数设置为8）

|  |  |  |
| --- | --- | --- |
|  | N=20000 | N=40000 |
| 使用omp处理 | 0.075490s | **0.262119s** |
| 不使用omp处理 | **0.072416s** | 0.283344s |

原因可能是，首先这里这部分计算并不是整个程序的时间瓶颈，其次这里的耗时本来很低，乘以一个1/N的系数也不会改变太多，相反创建线程和销毁线程需要一定的时间，因此对整个程序的效率提升不明显。下面的测试中并没有采用这种方式。（计算向量范数的时候仍然按要求采用了omp的并行方式）

**Prog3\_5:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | 矩阵模块 | 范数模块 | 加速比 | 效率 |
| 进程数=2 | N=20000 | 1.97649s | 0.00387s | 0.13543 | 0.0677 |
| N=40000 | 7.63473s | 0.00398s | 0.15381 | **0.0769** |
| N=80000 | 32.85739s | 0.00534s | 0.13638 | 0.0682 |
| 进程数=4 | N=20000 | 1.54947s | 0.00492s | 0.20681 | 0.0517 |
| N=40000 | 7.02763s | 0.00285s | 0.18290 | 0.0457 |
| N=80000 | 49.40616s | 0.00481s | 0.12566 | 0.0314 |
| 进程数=8 | N=20000 | **0.99049s** | **0.00273s** | 0.29928 | 0.0374 |
| N=40000 | **3.54806s** | 0.00287s | **0.33448** | 0.0418 |
| N=80000 | **18.24172s** | 0.00413s | 0.29598 | 0.0370 |

**Prog3\_6:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | 矩阵模块 | 范数模块 | 加速比 | 效率 |
| 进程数=4 | N=20000 | 1.90484s | 0.00470s | 0.14809 | 0.0370 |
| N=40000 | 6.40024s | 0.00271s | 0.20451 | **0.0511** |
| 进程数=9 | N=20001 | 1.5583s | 0.00260s | 0.16725 | 0.0186 |
| N=40002 | 5.49618s | 0.00292s | 0.23037 | 0.0256 |
| 进程数=16 | N=20000 | **1.49962s** | **0.00223s** | 0.18285 | 0.0114 |
| N=40000 | **5.33463s** | 0.00329s | **0.24647** | 0.0154 |

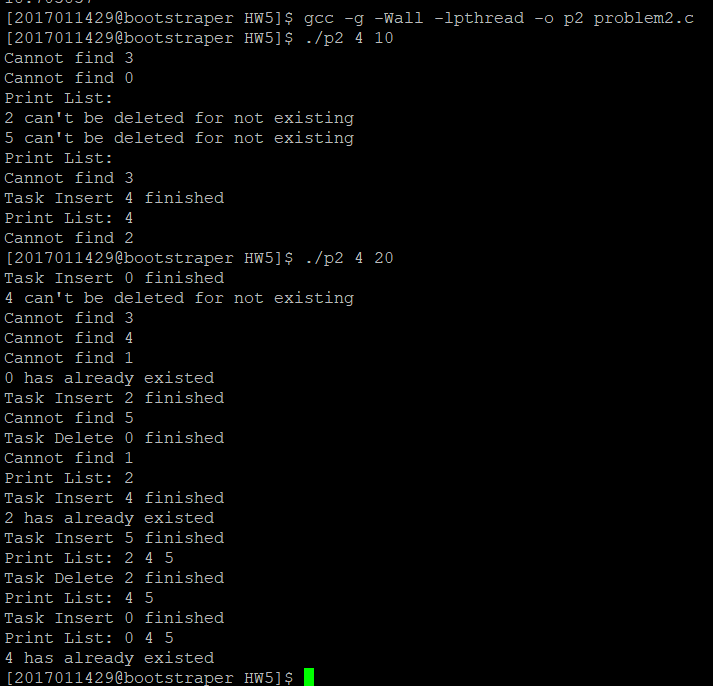
可以看到，由于分发时间的制约，进程数对运算时间模块的提升几乎没有太大作用，这也导致了效率值较高的通常是进程数较小的测例。

**PA 4.5**

本题使用两个链表分别作为任务队列和实际链表，实际链表中的元素均为有序，并且保证链表内没有相同元素（否则算插入失败）。程序首先生成数据，每当生成一个任务，就向任务队列里插入一个元素记录任务内容，并在线程等待数大于0时采用条件变量唤醒一个线程。与此同时，所有线程进入work开始工作，进入getdata函数，发现被pthread\_cond\_wait(&cond, &condmutex)卡住，因此所有线程都在这里暂时等待。每当主线程生成任务并修改条件变量，就使得一个线程得以跳出while循环并进行操作。当所有任务生成完毕后，程序广播条件变量，使得所有线程能够跳出循环并结束。

整个程序除了需要条件变量之外，还需要3个锁以及两个标志用的整数变量flag和waiting thread。下面分别解释这些量的作用。Mutexlist是链表锁，确保修改链表时只有一个线程在进行操作；condmutex防止多个线程同时申请pthread\_cond\_wait，保证每次只唤醒一个线程进行操作；Qmutex是对任务队列加锁，最开始我没有使用，因为每次获取任务的只能是一个线程，但后来发现Qmutex也是有用的，它可以防止主线程生成任务和其他线程获得并删除任务产生冲突。Flag标志全部任务是否均生成完毕；waiting thread记录正在等待唤醒的线程数，只有当等待数大于0时才能唤醒线程。

为了保证生成、删除操作能够大概率命中，我设定整个有序链表中最大的数字不超过6。在插入、删除以及寻址时成功或失败都会给出相应的提示信息。在4线程、300000的数据量下，计入所有输出所消耗的时间，程序用时约为10.3474s。下面是正确性截图，给出了两个测试例子。



**PA 5.3**

本题在主函数中实现了3个版本的排序，分别为串行计数排序、Qsort以及并行计数排序，并分别衡量它们的用时。实现的排序均为最朴素版本的排序，因为即便是分块计数然后再归并也用到了O(nlogn)的算法——这样意义不大，如果这样可以为什么不直接使用O（nlogn）的算法进行排序呢？所以这里只针对计数排序进行并行优化。

a)：i, j, count应该是私有变量，a, n, temp是公有变量。

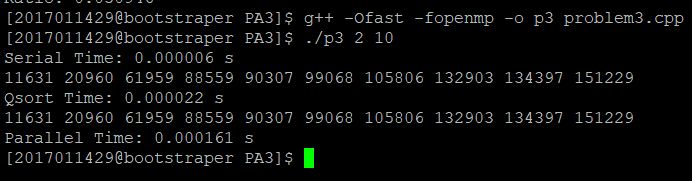
b)：没有循环依赖，因为每个线程执行的任务都是独立的，像temp这样的变量也只是每个线程各自修改其中的一部分。

c)：理论上可以，因为我们可以在外面套一个循环，将数组划分为几块，每块分别进行memcpy。重写memcpy代码当然理论上也可行。

e)：比串行的计数排序效果要好很多，比库函数效率要低，毕竟库函数的复杂度远远低于计数排序。具体比较效果如下。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | N=40000 | N=80000 | N=120000 | N=200000 |
| 库函数快排 | 0.003698s | 0.00772s | 0.012081s | 0.020698s |
| 串行计数排序 | 4.019379s | 16.1676s | 35.92192s | 99.05818s |
| 线程数为2 | 2.030096s | 8.13316s | 17.88181s | 48.40773s |
| **加速比（2）** | **1.9799** | **1.9879** | **2.0088** | **2.0463** |
| 线程数为4 | 1.105275s | 4.254214s | 8.961174s | 26.39524s |
| **加速比（4）** | **3.6365** | **3.8004** | **4.0086** | **3.7529** |
| 线程数为8 | 0.618403s | 2.29396s | 4.959635s | 13.61723s |
| **加速比（8）** | **6.4996** | **7.0479** | **7.7092** | **7.2745** |

可见，对于串行计数排序而言，并行计数排序基本达到了线性加速比的效果。正确性截图如下图，供参考。



**PA 4**

本题前两问要求用omp的静态调度和动态调度测试给定程序的并行效率，代码中提供了static\_omp以及dynamic\_omp函数，分别实现了静态调度和动态调度的omp程序。选取N=50000，num\_threads=4，chunk\_size分别为1、20、400，对两个程序的运行时间进行测试，结果如下表。

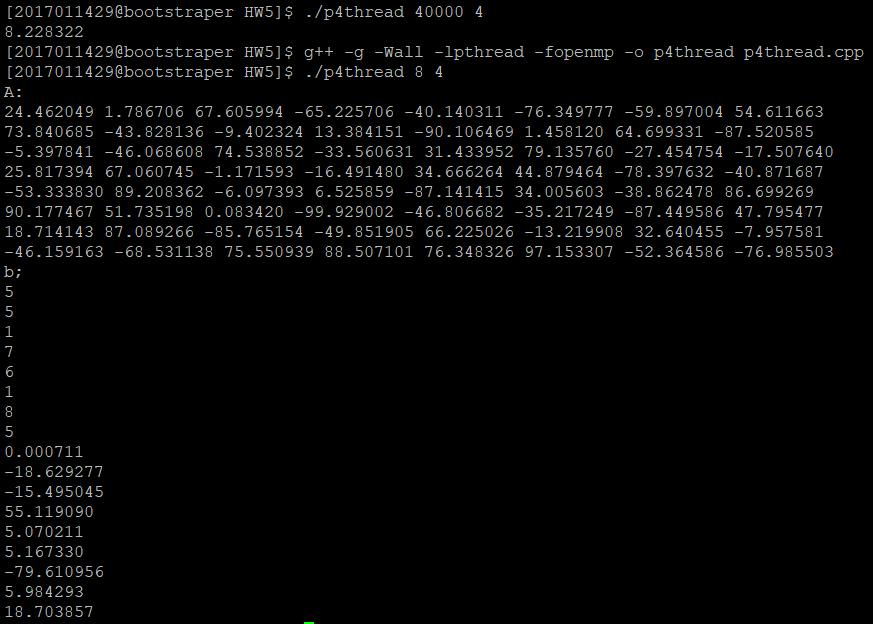
|  |  |  |  |
| --- | --- | --- | --- |
|  | Chunk\_size | 运行时间 | 加速比 |
| 串行程序 |  | 4.78287s |  |
| 静态调度方法 | 1 | 2.59860s | 1.8406 |
| 20 | 2.56464s | 1.8649 |
| 400 | 2.50927s | 1.9061 |
| 动态调度方法 | 1 | 1.24565s | 3.8396 |
| 20 | **1.17562s** | **4.0684** |
| 400 | 1.24336s | 3.8467 |

默认情况下，用时1.9590秒，加速比为2.4415，介于静态和动态之间。

第三问使用了pthread进行编程，借用了T2的任务队列，实现动态调度。可能由于T2的实现是基于链表，所以整体效率不够高，以下是几轮测试，线程数都为4。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | N=10000 | N=20000 | N=40000 | N=50000 |
| Time | 0.48029s | 2.26253s | 8.22832s | 12.38072s |

以下是pthread程序正确性截图，供参考。注：第一问的命令行参数多了chunk\_size；两道小问的程序的命令行参数中，线程数和问题规模与前面三道题相反。



（压缩包内的所有程序都已经上传到集群中。PA终于结束啦~~完结撒花~ 助教辛苦了！其实写这个作业也挺辛苦的，因为总是出现很玄学的问题（比如用clock计时慢很多导致看不出效率提升之类的）无法解决\_(:з」∠)\_然后每天缠着75的llx和73的zl问\_(:з」∠)\_ 在最后感谢助教和这两位同学吧！）