# 多线程并行快速排序算法

单暖乔 21307130395

我使用的是OpenMP的方式完成并行算法的，是在vscode中用的c++语言和omp.h完成的

## 1. 文件说明和环境配置

这个作业我有源文件quicksort_compare.cpp和对应的可执行文件。

这里我直接在vscode中的c++环境中实现，每次编译运行的时候要在terminal输入g++ -fopenmp {文件路径}/{文件名}.cpp -o {文件名} 并且输入.\{文件名}.exe就可以运行

## 2.代码思路

这里我用了课上讲的这个算法作为我整体代码的逻辑

```
unsigned __stdcall QuickSort(qSortIndex *m)
{
   int p = m->lo;
   int r = m->hi;
   if (p < r)
   {
     qSortIndex s, t;
     HANDLE tH[2];
     int q = Partition(p, r);
     s.lo = p; s.hi = q-1;
     tH[0] = (HANDLE)_beginthreadex (NULL, 0, QuickSort, &s, 0, NULL);
     t.lo = q+1; t.hi = r;

     tH[1] = (HANDLE)_beginthreadex (NULL, 0, QuickSort, &t, 0, NULL);
     WaitForMultipleObjects(2, tH, INFINITE);
   }
   return 0;
}
```

就是每次进行划分之后要开启两个子线程，递归执行直到左边界大于等于右边界。这里我使用的方法是用#pragma omp parallel和#pragma omp single和 #pragma omp task的方式产生子线程的，（因为怕sections的具体实现比如同步障和资源分配会影响到整个的性能，自己尝试使用的时候确实慢），其余的逻辑和串行的是一样的（就是去掉pragma注释就是串行程序）。之后我也写了串行的sort，便于对运行时间的比较。因为快速排序的时间和原始序列的元素的大小顺序有关系，为了能得到比较稳定的效果，我每次运行的时候会迭代15次，以观察平均效果。

## 3.代码说明

我定义了三个函数

step函数是做quicksort的partition用的。

quicksort函数是调用partition并进行递归的，这里在递归的时候要用到并行的逻辑。

quicksort_serial函数就是串行版本的quicksort，这里只需要去掉并行的注释。

之后在main函数中实现对参数的设置，包括总数据量N，迭代次数iter和线程数（用的是omp_set_num_threads）；并计算运行时间，进行性能比较。

## 4.实验结果

这些都是按30次迭代取平均得到的，分别对2个线程，4个线程，8个线程的情况下的不同数据量时的运算结果。虽然已经三十次的重复实验了，但是由于每次运行的具体情况不同，在同一个数据量的时候串行的时间可能也会出现些许的出入。

**2个线程**

| 数据量 | 加速比 | 串行平均执行时间/s | 并行平均执行时间/s |
|---|---|---|---|
| 1K | 0.272704 | 9.99928e-005 | 0.000366672 |
| 5K | 0.482752 | 0.000466657 | 0.00096666 |
| 10K | 0.58695 | 0.00089999 | 0.00153333 |
| 100K | 1.34763 | 0.0104667 | 0.00776673 |
| 1M | 1.3604 | 0.108333 | 0.0796333 |
| 10M | 1.36579 | 1.94233 | 1.42213 |
| 100M | 1.74522 | 99.2576 | 56.8739 |

**4个线程**

| 数据量 | 加速比 | 串行平均执行时间/s | 并行平均执行时间/s |
|---|---|---|---|
| 1K | 0.115375 | 9.99928e-005 | 0.000866675 |
| 5K | 0.207545 | 0.000366664 | 0.00176667 |
| 10K | 0.451598 | 0.000933321 | 0.00206671 |
| 100K | 1.63541 | 0.0104666 | 0.00640002 |
| 1M | 2.14212 | 0.111033 | 0.0518333 |
| 10M | 2.58081 | 1.95677 | 0.7582 |
| 100M | 3.20442 | 94.3795 | 29.4529 |

**8个线程**

| 数据量 | 加速比 | 串行平均执行时间/s | 并行平均执行时间/s |
| --- | --- | --- | --- |
| 1K | 0.0263141 | 3.33309e-005 | 0.00126665 |
| 5K | 0.11765 | 0.000400011 | 0.0034 |
| 10K | 0.190476 | 0.000799998 | 0.0042 |
| 100K | 1.05595 | 0.0100667 | 0.00953333 |
| 1M | 2.18588 | 0.1125 | 0.0514667 |
| 10M | 3.16402 | 1.94387 | 0.614367 |
| 100M | 5.2668 | 90.22 | 17.1299 |

# 5.结果分析

在上述的实验中，可以看到当数据量的时较小的时候。线程数越多反而在执行上耗时更长。但是数据量大的时候（在1M及之后），线程数的增加会减少执行的时间。

并且在数据量小于等于10K的时候，加速比小于1也就是说这个时候并行的时间要比串行的时间长。数据量大于10K的时候，并行的程序才显示出优势。

上述现象产生的原因应该是类似的。毕竟可以把串行的程序看成是线程数为1的情况。

在数据量比较大的时候线程越多，并行度更大，减少执行的时间。而在数据量比较小的时候，并行执行对时间的节省是抵不过额外开销的，线程的数量容易过多，线程的创建终止，调度的额外开销就会显现出来。

这个程序中，线程是在运行的时候动态创建的，在计时里。且在每个线程执行划分之后再创建线程的，这个过程是在串行进行的，所以并行度也会受到影响。因此在数据量比较小的时候，相对来说就难以用更多的线程达到好的效果。

# 6.总结

感受和评价：

这个实验的环境相对来说比较好配。算法的思想比较明晰，尝试了对OpenMP的使用。这个试单的代码上，可以看到如果将并行的注释去掉，这个就是正常的串行的quicksort的代码。符合了OpenMP的编程理念。

关于加速比有这些结论：

在固定线程数量的时候，当数据量增大的时候，加速比会增大。

在一定的范围之内（和数据量有关，数据量越大，范围越宽泛），加速比会随着线程数量的增多而增大。