分布式与并行计算 课程Project

201300096 杜兴豪

本次Project需要我们实现快速排序、归并排序、枚举排序的串行和并行算法。

并行算法实现

Merge Sort

按照课本上的PSRS排序技术，算法的执行过程是这样的：

给定所要排序的数组A[1,2,...n]和所要准备使用的处理器数p：

- 将数组A[1,...n]均匀分成p段，每个处理器分到子数组A[(i-1)n/p+1,...inp]
- 这p个处理器（体现为p个线程）调用串行的排序算法把分配到的子数组段排序，并在每个排好序的子数组中选出p个样本放到共享的主元数组中
- 再分配一个线程对刚刚赋好值的主元数组调用串行算法排序，从中选出p-1个主元，并存入共享的最终主元数组中，以便于前p个线程访问
- 每个线程获得信号后，按照最终的主元数组将分配到的子数组划分成至多p段
- 划分结束后，每个线程将对应的段按照段号交换到对应的线程中
- 全部交换完毕之后，每个线程对发送到自己的所有数据段进行归并排序
- 排序完成后，由刚才用于排序主元数组的线程对这些有序段进行综合拼接。

算法伪代码

用MergeThread代表分配数组段的线程，用MainThread代表排序主元并执行最终拼接任务的线程。算法的伪代码为：

```
function Parallel_MergeSort(Array, 1, n):
    create_and_run(MainThread)
    for i=1 to p:
        create_and_run(MergeThread(Array, (i-1)n/p+1, ni/p))

Thread MergeThread(Array, head, tail):
    task1:
        serial_sort(Array, head, tail)
        pivotArray <- pick_pivots(Array, head, tail)
    task2:
        divide(Array, head, tail)
        broadcast to other MergeThreads
    task3:
        sort items received
    do:
        task1
        wait for MainThread.task1
        task2
        wait for all MergeThread.task2
        task3

Thread MainThread():
    task1:
```

```
        finalPivotArray <- sort_and_select_pivots(pivotArray)
    task2:
        merge all results of MergeThreads
    do:
        wait for all MergeThread.task1
        task1
        wait for all MergeThread.task3
        task2
```

技术要点

从伪代码中可以发现，这个方法效果放在一边不谈，实现起来就非常有难度。伪代码中涉及大量的互相等待任务：

1. 在所有MergeThread完成给主元数组赋值之前，MainThread不能开始排序主元数组（此时pivotArray中还没有初始值，是分配时的随机值）
2. 在MainThread完成主元数组的排序并确定最终的主元之前，MergeThread不能开始对子数组进行划分
3. 在所有MergeThread完成全局交换之前，任何MergeThread不能开始排序所接收到的数据（否则可能数据不完全）
4. 在所有MergeThread完成最终的排序任务之前，MainThread不能开始整理整个数组

为了尽可能正确地实现出PSRS算法，我在代码中调用了很多全局的信号量，来实现一等多，多等多的任务，并且也用信号量来模拟互斥锁，来实现多等一的任务。并且，由于线程之间抢占资源顺序的不确定性，我需要利用到多维可变长数组Vector<Vector<Integer>>来存储全局交换时的数据段。虽然最后成功实现出了PSRS算法，但是由于变长数组存取的时间复杂度都很高，信号量PV操作也很耗时，仅凭我的Java水平也难以优化我的实现，因此我最后的并行归并排序的运行效果并不好，是串行算法的数倍。虽然我的实现效果不尽人意，但是这同样也成为了我的优势———相信没有多少人成功实现出了PSRS算法，因此我坚持使用它作为我的并行归并排序。

Quick Sort

并行快排的思路十分简单：在串行算法中，我们选出主元后，需要串行地先对左子数组进行快排，再对右子数组进行快排，最终才能得到我们希望的结果。那么如果把分别排序左右子数组的过程变为并行，那时间开销就会降低很多。具体到实现中，就是分别为左右子数组的排序创建一个线程，让它们并行运行即可。

伪代码

```
function Parallel_QuickSort(Array):
    create_and_run(QuickThread(Array, 1, n))

Thread QuickThread(Array, head, tail):
    pivot = serial_partition(Array, head, tail)
    create_and_run(QuickThread(Array, head, pivot-1))
    create_and_run(QuickThread(Array, pivot+1, tail))
```

技术要点

这里的划分和排序全部都需要是Inplace的，如果不是的话，额外创建数组需要更多的存储空间，最终还需要多一步归并，进一步增大时间开销。

Enum Sort

并行枚举排序，按照同样的思路，在串行枚举排序中时间开销最大的是每个数的rank值的计算，因此把这个部分并行起来，整个算法就会快很多。注意到串行算法中，还需要知道每个数出现的次数，而在rank的计算过程中可以同时获得这个值，因此一并并行处理。

伪代码

```
function Parallel_EnumSort(Array):
    rank, count = [], []
    for i=1 to n:
        create_and_run(EnumThread(Array, i))
    create_and_run(MainThread(Array))

Thread EnumThread(Array, i):
    for id=1 to n:
        if(Array[id] < Array[i]):
            rank[i] += 1
        else if(Array[id] == Array[i]):
            count[i] += 1

Thread MainThread(Array):
    wait for all EnumThreads to finish
    serial_EnumSort(Array) using count and rank
```

技术要点

在ppt中只讲到可以将计算每个数的rank这里并发进行，然而没有提到当有重复数出现时的做法，也就是每个数的count的计算。对每个数而言，计算rank需要遍历整个数组一遍，也就是O(n)的，而计算count也需要遍历数组一遍。因此如果只并发执行rank而不并发count的话，整个算法的时间开销将不会变小。而count和rank实质上是做一样的事情，把当前数值和整个数组进行比较，因此可以同时完成，这样就能极大减小时间开销。

运行时间比较

	串行	并行
MergeSort	2824160ns	51920170ns
QuickSort	2884970ns	533380ns
EnumSort	890408610ns	6425334760ns

分析实验结果

QuickSort的并行性能有很大提高，然而MergeSort和EnumSort的表现不尽如人意。对代码内部原理进行分析可知，并行的MergeSort和EnumSort中用到了很多信号量，而QuickSort中并没有使用到信号量。因此为了优化运行时间，可以尝试在未来实现中尽量避免使用过多的信号量。此外，算法效率低的另一可能是我在代码执行过程中使用了存取时间复杂度较长的容器vector（见MergeSort），算法的实现应当尽可能精简，我还需要再努力。

代码结构：

- parallel.java：存放了parallel类以及它所属的三种并行算法类：
    - Parallel_MergeSort_1类：采用PSRS算法实现的并行算法
    - Parallel_MergeSort_2类：仅仅将串行归并排序的排序部分并行化
    - Parallel_QuickSort类
    - Parallel_EnumSort类
- series.java：存放series类以及其所属的三种串行排序算法：
    - MergeSort
    - QuickSort
    - EnumSort

具体运行方法见ReadMe.txt