分布式与并行计算 课程实验报告

201300066 麻超

实验要求

- 利用Java或者C#多线程模拟并行处理，实现快速排序、枚举排序、归并排序三种排序方法的串行和并行算法，数据集中共30000个数据，范围为[-50000,50000].
- 说明程序执行方式，记录在ReadMe.md（即本文件）中。
- 读取乱序数据文件random.txt，排序完成后输出排序文件order.*txt*。 *(需提交六份order*.txt，命名为order1.txt，order2.txt...以此类推)
- 比较各种算法的运行时间，请将运行时间记录在2*3的表格中。行分别表示串行、并行，列分别表示快速排序、枚举排序、归并排序。（每个排序算法进行五次实验，求平均值）
- 撰写实验报告，包括并行算法的伪代码、运行时间、技术要点（如性能优化方法）等，结合各自的实验设备（如多核处理器）上的实验结果进行优化，并在实验报告中针对实验结果进行分析（考虑到并行算法多线程在单核处理器中的并行开销，有可能性能会比串行算法下降）。
- 独立完成实验，杜绝抄袭。
- 实验使用的设备为四核心/八线程，实验环境为Java 17(version 19.0.1).

算法描述

快速排序（串行）

是一种经典的时间复杂度为$O(n\log_2 n)$的排序方法，该方法的基本思想是从数列中取出一个数作为基准，再将整个数组中所有比它小的放到它的左边，比它大的放到它的右边，得到两个分数组，再对两个分数组重复执行该过程，直到每个区间只有一个数。

伪代码如下：

```
function quick_sort(arr,left,right):
    if left<right:
        temp=partition(arr,left,right)
        quick_sort(arr,left,temp-1)
        quick_sort(arr,temp+1,right)
    end if
end function
function partition(arr,left,right):
    temp=arr[left]
    while(left<right){
        while(left<right && a[right]>temp):
            right-=1
        end while
        if(left<right):
            a[left++]=a[right]
        end if
        while(left<right && a[left]<=temp):
                left++
        end while
        if(left<right):
            a[right--]=a[left]
        end if
```

```
    end while
    a[left]=temp
    return left
end function
```

经过十次实验，快速排序（串行）的运行时间约为8.7ms.

归并排序（串行）

和快速排序一样，其时间复杂度为$O(n\log_2 n)$.归并排序主要体现分而治之的思想，先将数组分为两个大小相等的分数组，然后对这两个分数组（已排好序的）按照从小到大的顺序归并，对每个分数组都要进行这样的操作。伪代码如下：

```
function merge_sort(arr,left,right):
    if(right>left):
        mid=(left+right)/2//mid为分开数组的点
        merge_sort(arr,left,mid)
        merge_sort(arr,mid+1,right)
        merge(arr,left,mid,right)
    end if
end function
function merge(arr,left,mid,right):
    len=right-left+1
    new arr1[len]
    index=0
    while(left<=mid && mid+1<=right)
        if(arr[left]<arr[mid+1]):
            arr1[index]=arr[left]
            left++
        else:
            arr1[index]=arr[mid+1]
            mid++
        end if
        index++
    end while
    while(left<=mid):
        arr1[index]=arr[left]
        left++
        index++
    end while
    while(mid+1<=right):
        arr1[index]=arr[mid+1]
    end while
    copy arr[1] to arr[left:right]
end function
```

经过十次实验，归并排序（串行）的运行时间约为6.9ms.

枚举排序

与前二者不同，枚举排序的时间复杂度为$O(n^2)$.这就体现出其运行时间与前二者有较大差异。算法思想为对数组中每个元素，统计有多少个比它小的数，得到它在数组的位次，再按照每个元素的位次得到新数组。由于给定的数据集没有重复数据，所以这里我没有考虑有元素相等的情况。伪代码如下：

```
function rank_sort(arr):
    new arr1[]
    for i in arr:
        k=0
        for j in arr:
            if arr[i]>arr[j]:
                k++
            end if
        arr1[k]=arr[i]
        end for
    end for
    return arr1[]
end function
```

经过十次实验，枚举排序（串行）的运行时间约为692.1ms.

快速排序（并行）

并行化快速排序的核心是将数组分为两组之后，将左边的部分和右边的部分分别交给不同的进程处理，如此递归进行。

```
function ParallelQuickSort(arr,left,right):
    p = partition(arr, left, right)
    pi send arr[p + 1 : right] to pi+1
    ParallelQuickSort(L, left, p - 1)
    ParallelQuickSort(L, r + 1, right)
    pi+1 send L[p + 1 : right] to pi
end function
```

经过十次实验，快速排序（并行）的运行时间约为12.6ms.

归并排序（并行）

归并排序的并行化主要是将递归的过程分别交给不同的进程，最后Merge的部分交给串行函数处理。

```
function ParallelMergeSort(arr,left,right):
    m = (start + end) / 2
    if right-left>=1:
        Pid+1 do ParallelMergeSort(L, left, m)
        Pid+1 do ParallelMergeSort(L, m + 1, right)
    end if
    merge(arr,left,mid,right)
end fucntion
```

经过十次实验，归并排序（并行）的运行时间约为26.3ms.

枚举排序（并行）

```
function ParallelRankSort(arr, n):
    P0 send L to P1, P2 ...,Pn
    for all Pi where 1<= i <= n para-do:
        count = 1
        for j = 0 to n do:
            if L[i] > L[j] || (L[i] == L[j] && i > j):
                 k = k + 1
            end if
        end for
    end for
    P0 merge the result
end function
```

经过十次实验，枚举排序（并行）的运行时间约为112.5ms.

具体实现

在用Java实现的过程中，我主要利用了Fork/Join框架维护线程池。Fork/Join框架是Java提供的一个用于并发执行任务的框架，其主要思想就是把大任务分割成若干的小任务，最终汇总每个小任务结果后得到大任务结果的框架。一般配合可分解任务接口ForkJoinTask使用，在实现过程中，我使用了Recursive Action抽象类来实现，因为其比较方便。以下为并行快速排序和并行归并排序的调用接口：

```
public static void parallel_quick_sort(int[] arr) {
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    ParallelQuickSort task = new ParallelQuickSort(arr, 0, arr.length-1);
    Future<Void> result = forkJoinPool.submit(task);
    try {
        result.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

此为并行化枚举排序的调用接口。

```
public static int[] parallel_rank_sort(int[] arr) {
    int[] re = new int[arr.length];
    final ForkJoinPool forkJoinPoolEsp = new
ForkJoinPool(Runtime.getRuntime().availableProcessors());
    forkJoinPoolEsp.invoke(new ParallelRankSort(arr, 0, arr.length-1, -1,
re));
    return re;
}
```

以下为并行快速排序的主要实现：

```
@Override
protected void compute() {
    int pivot = QuickSort.partition(arr, left, right);//调用串行程序中的
partition函数
    ParallelQuickSort task1 = null;
    ParallelQuickSort task2 = null;
    if (pivot - left > 1) {
```

```
        task1 = new ParallelQuickSort(arr, left, pivot-1);
        task1.fork();
    }
    if (right - pivot > 1) {
        task2 = new ParallelQuickSort(arr, pivot+1, right);
        task2.fork();
    }
    if (task1 != null && !task1.isDone()) {
        task1.join();
    }
    if (task2 != null && !task2.isDone()) {
        task2.join();
    }
}
```

以下为并行归并排序的主要实现：

```
@Override
protected void compute() {
int mid = (left + right)/2;
if (right - left >= 1) {
ParallelMergeSort mergeSortTask1 = new ParallelMergeSort(array, left, mid);
ParallelMergeSort mergeSortTask2 = new ParallelMergeSort(array, mid+1,
right);
mergeSortTask1.fork();
mergeSortTask2.fork();
mergeSortTask1.join();
mergeSortTask2.join();
}
MergeSort.merge(array, left, mid, right);//调用串行程序中的merge函数
}
```

在并行化快速排序和归并排序的过程中，我都选择了让其调用串行部分的对应函数。

以下为并行枚举排序的主要实现：

```
@Override
protected void compute() {
    if (index == -1) {
        List<ParallelRankSort> futures = new Vector<>();
        for (int i = start; i <= end; i++) {
            final ParallelRankSort newTask = new ParallelRankSort(arr,
start, end, i ,result);
            futures.add(newTask);
        }
        invokeAll(futures);
    }
    else{
        int k = 0;
        for (int j : arr) {
            if (arr[index] > j)
                k++;
        }
        result[k] = arr[index];
    }
```

```
19 }
```

实验分析

三种排序算法的并行/串行运行时间报告如下表所示：

	快速排序	归并排序	枚举排序
串行	8.7ms	6.9ms	692.1ms
并行	12.6ms	26.3ms	112.5ms

可以看到，并行化处理之后，快速排序和归并排序的运行时间并没有得到提升，这可能是由于在并行处理的过程中产生的通信、创建新进程等开销远大于实际节省的这一点时间，且归并排序相比于快速排序的开销更大，因为相较于快速排序，归并排序更依赖于$merge$函数，所以对串行的部分需求更高，导致了时间开销更大。而枚举排序在经过了并行化处理之后，其运行效率提高了84%，这一方面是由于枚举排序本来的运行效率过低，所以并行处理之后其效率提升就会更加明显，但是由于该算法本身的硬伤，导致其提升了效率之后与前两者仍然有着较大的差距。

为了体现出并行计算带来的效率提升，我又自建了一个大小为300000的随机数据集和一个大小为3000000的随机数据集，运行结果如下所示：

	快速排序	归并排序	枚举排序
串行	40ms	41ms	68495ms
并行	57ms	118ms	5224ms

上表：大小为300,000的数据集运行结果

	快速排序	归并排序	枚举排序
串行	269ms	428ms	/
并行	109ms	172ms	/

上表：大小为3,000,000的数据集运行结果

可以看出当数据量变大的时候，并行化的快速排序和归并排序在数据量较大的时候，其拥有更高的处理效率，在数据集为3,000,000的时候，二者与串行相比，提升了约一半的效率。在数据量为300,000时，枚举排序提升了约92%的效率，在数据量为3,000,000的时候，考虑到枚举排序那过于离谱的时间复杂度，所以我没有对其进行测试，但估计也在90%以上。

优化思路

针对以上实验过程中出现的问题，我考虑了以下几个可以改进的点：

- 在串行部分可以实现的改进有`QuickSort`的`Partition`函数，在数据结构与算法课程中我们学习到的可以提高快速排序运行效率的方法有选取基准时尽量选择中位数，使基准左右两边的数组大小接近，以更加接近理想的$O(n\log_2 n)$时间复杂度。
- 在并行部分，也可以考虑一些改进方法，例如针对特定的处理器进程数规定最大进程数，同时当问题规模比较小的时候，可以考虑不再分发给新进程，而是直接处理，以减少通信和创建、撤销进程所占用的时间开销。

- 当前情况下，所有线程都有着相同的优先级，这在一定程度上限制了问题的运行效率，所以可以考虑给每个进程加上不同的优先级进行处理。

其他

在整个项目中，我人为创建了三个文件夹，以`num_30000`,`num_300000`,`num_3000000`命名，分别代表了实验所给的数据集和我自己建立的数据集的实验情况。如果需要进行实验，要在`ParallelCompute.java`文件第7行中修改数组最大容量，并将对应的数据集`random`文件拖到项目一级文件夹下。

本次实验并不复杂，但让我学到了很多知识，了解了并行化处理的基本过程并投入了实践之中，向老师和各位助教表示感谢。