分布式与并行计算实验报告

实验：三种排序的串行与并行比较

院系：人工智能学院

姓名：方盛俊

学号：201300035

邮箱：201300035@smail.nju.edu.cn

时间：2022年11月28日

目录

一、项目要求

- 课程Project：三种排序的串行与并行比较；
- 任务描述：请分别实现快速排序、枚举排序、归并排序三种排序方法的串行与并行算法。
- 数据集：random.txt，一共有30000个乱序数据，数据范围是[-50000, 50000]，数据间以空格 “ ” 分隔。

具体要求：

1. 用 Java 多线程或者 C# 多线程模拟并行处理（推荐用Java）。
2. 说明程序执行方式，记录在 `ReadMe.txt` 中（PS：这里我使用 `README.md` 替代，同时也作为实验报告的源文件）。
3. 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）
4. 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。
5. 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）。
6. 独立完成实验，杜绝抄袭。

执行方式：

在 Java 8 或以上的版本，使用 VS Code 及插件 Extension Pack for Java 的环境。

执行 `src/Test.java`，输入位于 `input/random.txt`，输出位于 `output/random/order*.txt`。

二、伪代码

2.1 快速排序 - 串行

```
def partition(data, k, l):
    pivot = data[l]
    i = k - 1
    for j in range(k, l):
        if (data[j] <= pivot):
            i += 1
            swap(data[i], data[j])
    swap(data[i + 1], data[l])
    return i + 1

def quicksort(data, i, j):
    if (i < j):
        r = partition(data, i, j)
        quicksort(data, i, r - 1)
        quicksort(data, r + 1, j)
```

2.2 快速排序 - 并行

```
def partition(data, k, l):
    pivot = data[l]
    i = k - 1
    for j in range(k, l):
        if (data[j] <= pivot):
            i += 1
            swap(data[i], data[j])
    swap(data[i + 1], data[l])
    return i + 1

def para_quicksort(data, i, j):
    if (i < j):
        r = partition(data, i, j)
        task1 = para_quicksort(data, i, r - 1)
        task2 = para_quicksort(data, r + 1, j)
        invoke(task1, task2)
```

2.3 枚举排序 - 串行

```
def ranksort(data):
    n = len(data)
    for i in range(n):
        k = 0
        for j in range(n):
            if (data[i] > data[j]) or (data[i] == data[j] and i > j):
                k += 1
        sorted_data[k] = data[i]
```

2.4 枚举排序 - 并行

```
def para_ranksort(data):
    n = len(data)
    P0 send L to P1, P2, ..., Pn
    for all Pi where 1 <= i <= n para-do:
        k = 0
        for j in range(n):
            if (data[i] > data[j]) or (data[i] == data[j] and i > j):
                k += 1
        sorted_data[k] = data[i]
```

2.5 归并排序 - 串行

```
def merge(data, k, m, l):
    left_data = data[k: m+1]
    right_data = data[m+1: l+1]
    i = 0
    j = 0
    for k in range(left, right + 1):
        if i == len(left_data):
            data[k] = right_data[j]
            j += 1
        elif j == len(right_data):
            data[k] = left_data[i]
            i += 1
        elif left_data[i] <= right_data[j]:
            data[k] = left_data[i]
            i += 1
        else:
            data[k] = right_data[i]
            j += 1


def mergesort(data, i, j):
    if (i < j):
        m = (i + j) // 2
        mergesort(data, i, m)
        mergesort(data, m + 1, j)
        merge(data, i, m, j)
```

2.6 归并排序 - 并行

```
def merge(data, k, m, l):
    left_data = data[k: m+1]
    right_data = data[m+1: l+1]
    i = 0
    j = 0
    for k in range(left, right + 1):
        if i == len(left_data):
            data[k] = right_data[j]
            j += 1
        elif j == len(right_data):
            data[k] = left_data[i]
            i += 1
        elif left_data[i] <= right_data[j]:
            data[k] = left_data[i]
            i += 1
        else:
            data[k] = right_data[i]
            j += 1


def para_mergesort(data, i, j):
    if (i < j):
        m = (i + j) // 2
        task1 = para_mergesort(data, i, m)
        task2 = para_mergesort(data, m + 1, j)
        invoke(task1, task2)
        merge(data, i, m, j)
```

三、Java 实现

Java 有着丰富的多线程库，这里我主要用了两种线程池：针对分治任务的 **ForkJoinPool** 与针对非分治任务的 **FixedThreadPool**。

3.1 ForkJoinPool 与 RecursiveAction

快速排序与归并排序是分治任务，如果我们手动维护分治任务的线程优先级的话，会十分复杂，我们需要让线程从分治任务线程树的底层（叶子节点）开始，一层一层地执行，直到最后执行到顶层（根节点），想要维护这样一个线程树的优先级，无疑是一个复杂的任务。

Java 为我们设计分治任务提供了一个十分友好的 API，也就是 `ForkJoinPool` 与 `RecursiveAction`。ForkJoinPool 是自 Java7 开始，由 jvm 提供的一个用于并行执行的任务框架。其主旨是将大任务分成若干小任务，之后再并行对这些小任务进行计算，最终汇总这些任务的结果，得到最终的结果。类似于单机版的 MapReduce，也是采用了分治算法，将大的任务拆分到可执行的任务，之后并行执行，最终合并结果集。

`RecursiveAction` 和 `RecursiveTask` 是其中的两种实现方式，其中 `RecursiveAction` 没有返回值，`RecursiveTask` 有返回值。由于我们准备在 **原数组** 上进行分治任务，所以我们就直接采用 `RecursiveAction` 的方式，忽略 `RecursiveTask` 了。

如果我们已经实现了一个串行版本的分治任务，我们可以将其很简单地便修改为并行版本。

以快速排序为例，串行版本的快速排序如下：

```
public class QuickSort {
    private static void _quickSort(
        List<Integer> numbers, int left, int right) {
        if (left < right) {
            int pivot = partition(numbers, left, right);
            _quickSort(numbers, left, pivot - 1);
            _quickSort(numbers, pivot + 1, right);
        }
    }

    public static List<Integer> quickSort(List<Integer> numbers) {
        List<Integer> sortedNumbers = new ArrayList<>(numbers);
        _quickSort(sortedNumbers, 0, sortedNumbers.size() - 1);
        return sortedNumbers;
    }
}
```

我们可以将其迅速地改为并行版本：

```
public class ParallelQuickSort extends RecursiveAction {
    public List<Integer> numbers;
    private int left;
    private int right;

    public ParallelQuickSort(
            List<Integer> numbers, int left, int right) {
        this.numbers = numbers;
        this.left = left;
        this.right = right;
    }

    @Override
    protected void compute() {
        if (left < right) {
            int pivot = partition(numbers, left, right);
            ParallelQuickSort leftTask =
                new ParallelQuickSort(numbers, left, pivot - 1);
            ParallelQuickSort rightTask =
                new ParallelQuickSort(numbers, pivot + 1, right);

            leftTask.fork();
            rightTask.fork();

            leftTask.join();
            rightTask.join();
        }
    }

    public static List<Integer> parallelQuickSort(
            List<Integer> numbers) {
        List<Integer> sortedNumbers = new ArrayList<>(numbers);
        ForkJoinPool pool = ForkJoinPool.commonPool();
        ParallelQuickSort task = new ParallelQuickSort(
            sortedNumbers, 0, sortedNumbers.size() - 1);
        pool.invoke(task);
        return task.numbers;
    }
}
```

由于需要重载的 `compute` 方法是没有参数的，因此我们通过 **构造函数** `ParallelQuickSort` 将参数保存在对象的 **成员变量** 里，然后再在 `compute()` 里调用。

在 `compute()` 里，我们通过 `task.fork()` 和 `task.join()` 对任务进行执行。在 `parallelQuickSort()` 里，我们通过 `ForkJoinPool.commonPool()` 创建了一个线程池，接着创建根任务 `task`，最后通过 `pool.invoke(task)` 执行。执行完毕后的结果保存在 `task.numbers` 里。

快速排序是这样，归并排序也是同理，这里就不过多赘述了。

3.2 FixedThreadPool 与 ExecutorService

相比于快速排序和归并排序，枚举排序并不是分治任务，因此我们使用更为基础的 `ExecutorService` 与 `FixedThreadPool`。

枚举排序的串行版本十分简单：

```
public class RankSort {
    public static List<Integer> rankSort(List<Integer> numbers) {
        List<Integer> sortedNumbers = new ArrayList<>(numbers);
        for (int i = 0; i < numbers.size(); i++) {
            int rank = 0;
            for (int j = 0; j < numbers.size(); j++) {
                if (numbers.get(j) < numbers.get(i)
                        || (numbers.get(j) == numbers.get(i) && j < i)) {
                    rank++;
                }
            }
            sortedNumbers.set(rank, numbers.get(i));
        }
        return sortedNumbers;
    }
}
```

我们将其修改为并行版本，也就是要将最外层的循环并行化：

```
public class ParallelRankSort implements Runnable {
    public List<Integer> numbers;
    List<Integer> sortedNumbers;
    private int i;

    public ParallelRankSort(
            List<Integer> numbers, List<Integer> sortedNumbers, int i) {
        this.numbers = numbers;
        this.sortedNumbers = sortedNumbers;
        this.i = i;
    }

    @Override
    public void run() {
        int rank = 0;
        for (int j = 0; j < numbers.size(); j++) {
            if (numbers.get(j) < numbers.get(i)
                    || (numbers.get(j) == numbers.get(i) && j < i)) {
                rank++;
            }
        }
        sortedNumbers.set(rank, numbers.get(i));
    }

    public static List<Integer> parallelRankSort(List<Integer> numbers) {
        List<Integer> sortedNumbers = new ArrayList<>(numbers);
        ExecutorService executor = Executors.newFixedThreadPool(8);
        for (int i = 0; i < numbers.size(); i++) {
            ParallelRankSort task =
                new ParallelRankSort(numbers, sortedNumbers, i);
            executor.submit(task);
        }
        executor.shutdown();
        try {
            if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException ex) {
            executor.shutdownNow();
            Thread.currentThread().interrupt();
        }
        return sortedNumbers;
    }
}
```

这里我们依然是要通过 **构造函数** `ParallelRankSort` 将参数保存在对象的 **成员变量** 里，以便重载的 `run()` 方法调用。

为了并行地进行外层循环，我们使用了 `Executors.newFixedThreadPool(8)` 创建了一个固定 8 个线程的（我的电脑的核心数为 8 个）的线程池对应的 `ExecutorService`。

我们通过 `executor.submit(task)` 将任务一个一个地加入到 `ExecutorService` 里，最后通过 `executor.shutdown()` 和 `executor.awaitTermination(60, TimeUnit.SECONDS)` 等待线程执行结束后，返回最后的结果 `sortedNumbers`。

四、技术要点

4.1 对比测试

为了避免排序算法出错，尤其是对并行化的排序算法进行检验，我们需要进行对比测试。

```
List<Integer> sortedNumbers = new ArrayList<>(numbers);
if (verify) {
    sortedNumbers.sort(Integer::compareTo);
}
if (verify) {
    System.out.println(
        "QuickSort - serial: " + sorted.equals(sortedNumbers));
}
```

如果 `verify == true` 的话，就会通过 Java 自带的排序算法进行排序，得到一个用于对比的排序后数据，然后通过 `sorted.equals()` 对内容进行对比。

通过这种方式，我们就可以保证及时地进行单元测试，保证算法没有错误。

4.2 阈值

分治算法会不断地将大任务分为小任务，但是如果我们分的小任务过小，就会创建太多的任务，会导致将大部分的计算资源消耗在创建线程和对任务分配的管理

上，导致出现效率下降的问题。

这时候我们可以对子任务大小进行判断，这里我设置了一个阈值 `THRESHOLD = 1024`，在子任务小于 `1024` 个数据时，直接进行串行的算法。

```
protected void compute() {
    if (right - left > THRESHOLD) {
        // para-sort ...
    } else {
        // serial-sort ...
    }
}
```

通过增加阈值的方式，并行的快速排序和归并排序的速度都有了不错的提升，例如归并排序就从 83 ms 减少到了 20 ms。

4.3 数据表示形式

本次实验我选用了 `List<Integer>` 来存储中间的排序数据，因此相对于 `int[]` 的存储形式，性能有些下降。

因为 `List<Integer>` 要在 `int` 和 `Integer` 之间进行装箱和拆箱，而且 `Integer.get(i)` 相较于 `int[i]` 的性能也会有点下降，因此执行较慢。

但是使用 `List<Integer>` 相较于 `int[]` 来说，支持的数据类型更为广泛，兼容性更好，因此这里我依然选用了 `List<Integer>`。如果后续需要进一步提升性能的话，可以将 `List<Integer>` 改为 `int[]`。

4.4 优化方向

- 快速排序的 `partition` 可以进一步地并行化，使得快速排序的速度进一步提升；
- 选用其他能够更好地并行化的算法，例如 DoubleMerge；
- 对不同的线程设定不同的优先级。

五、运行时间

5.1 `random.txt` 运行时间

我们多次执行 `Test.java`，以 `input/random.txt` 作为输入，得到的一个典型输出如下：

```
size of random.txt: 30000
QuickSort - serial: order1.txt
QuickSort - serial: 19 ms
QuickSort - serial: true
QuickSort - parallel: order2.txt
QuickSort - parallel: 19 ms
QuickSort - parallel: true
RankSort - serial: order3.txt
RankSort - serial: 3563 ms
RankSort - serial: true
RankSort - parallel: order4.txt
RankSort - parallel: 591 ms
RankSort - parallel: true
MergeSort - serial: order5.txt
MergeSort - serial: 36 ms
MergeSort - serial: true
MergeSort - parallel: order6.txt
MergeSort - parallel: 18 ms
MergeSort - parallel: true
```

我们取多次的输出求平均，最后得到的针对 30000 个数据的排序花费如下：

算法	串行	并行
快速排序 (QuickSort)	19 ms	20 ms
枚举排序 (RankSort)	3506 ms	560 ms
归并排序 (MergeSort)	32 ms	20 ms

5.2 `random10.txt` 运行时间

我们将 `random.txt` 复制成 10 行，将数据规模变为原来的 10 倍，也就是 300000 个数据。

我们取多次的输出求平均，最后得到的针对 300000 个数据的排序花费如下：

算法	串行	并行
快速排序 (QuickSort)	130 ms	77 ms
枚举排序 (RankSort)	391086 ms	60154 ms
归并排序 (MergeSort)	159 ms	109 ms

可以看出，在数据量更大的情况下，三种算法的并行算法均优于串行算法。

六、结果分析

6.1 快速排序

在数据量较小时，快速排序的串行算法和并行算法相差无几。

可能的原因如下：

- 快速排序是一个分治任务，进行并行化的时候要考虑父任务和子任务的优先级，不能直接地并行处理。
- Java 在创建子任务与子线程的时候，会花费大量资源在维护线程之间的通信与任务分配上，也就是并行额外开销较大。
- 数据量较小，因此并行的额外开销相对于进行的计算，就显得更大。
- 快速排序的 `partition` 仍然可以进一步地并行化。

因此我们通过增大数据规模，我们就可以看出，串行并行加速比有着明显的下降，从原来的 1 变成了几乎为 1/2。

6.2 枚举排序

枚举排序并不是一个分治任务，却是一个天生就适合分治的算法，我们可以很简单地将外层循环拆成不同的任务，然后交由多个线程并行处理。

我们计算加速比可知，30000 数据对应的加速比为 6.26，300000 数据对应的加速比为 6.50，两者都很接近我电脑的核心数 8 个。

但是 300000 数据对应的排序时间却增加了上百倍，我个人推测是硬件的缓存不足以容纳那么大的数据量，亦或者 Java 进行了垃圾回收。

6.3 归并排序

归并排序也是一种分治算法，和快速排序十分类似。但是归并排序在这里慢于快速排序，其中一个很重要的原因是，归并排序的 `merge` 总是要复制一个 `leftArray` 和一个 `rightArray`，而快速排序的 `partition` 是在原数组上进行的。

其他导致归并排序较慢的原因，也与快速排序差不多：

- 快速排序是一个分治任务，进行并行化的时候要考虑父任务和子任务的优先级，不能直接地并行处理。
- Java 在创建子任务与子线程的时候，会花费大量资源在维护线程之间的通信与任务分配上，也就是并行额外开销较大。
- 数据量较小，因此并行的额外开销相对于进行的计算，就显得更大。