实验报告

王科 201300008 scik@smail.nju.edu.cn

1 实验环境

CPU：使用4核8线程处理器，支持时分复用，操作系统统一调度。

语言：Java（jdk-11.0.12.7）

程序入口在 `.\src\Manager.java`

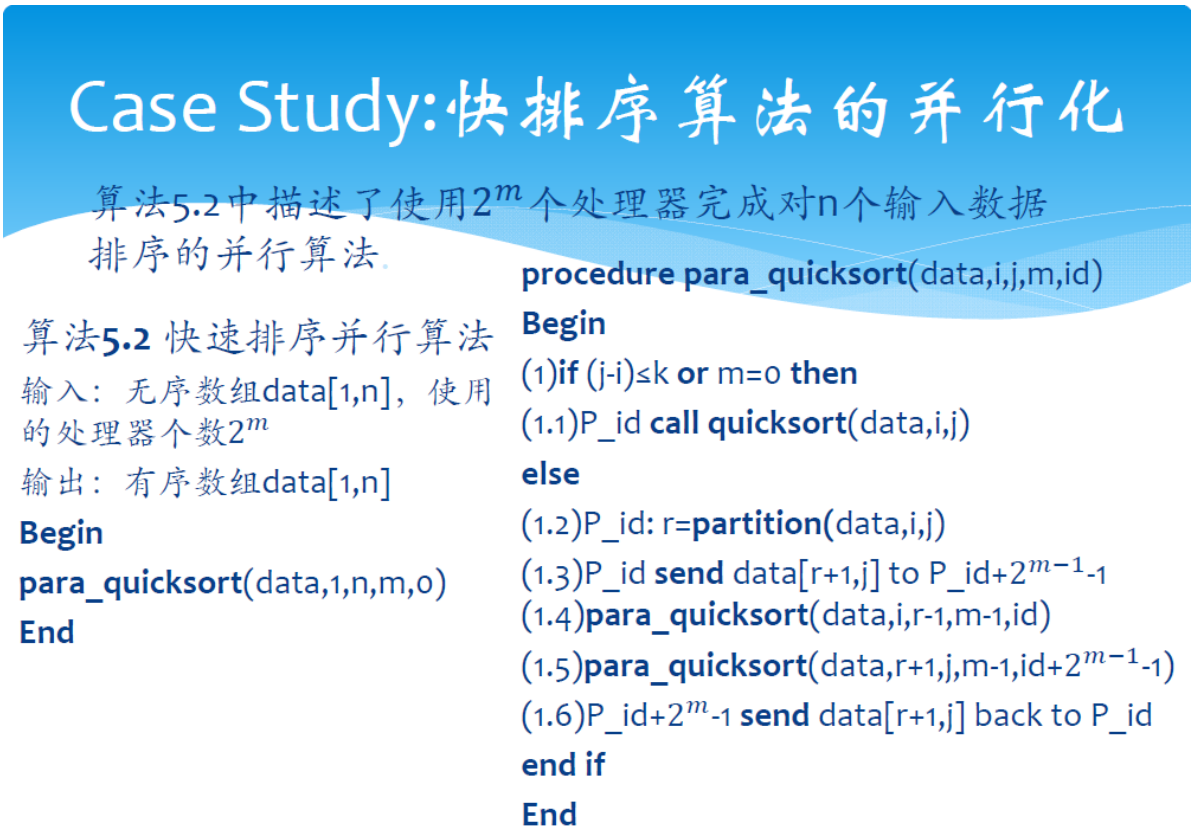
使用 `jdk-11.0.12.7-hotspot\bin\java.exe`

排序结果保存在 `.\results` 中

2 算法伪代码

2.1 快速排序

利用`partition`可以很自然地实现并行快速排序。



2.2 枚举排序

枚举的并行化是很简单的，只需每个处理器负责完成一部分元素的定位，然后将所有的定位信息集中到主进程中，由主进程负责完成所有元素的最终排位即可。

枚举排序的并行算法

对该算法的并行化是很简单的，假设对一个长为n的输入序列使用n个处理器进行排序，只需是每个处理器负责完成对其中一个元素的定位，然后将所有的定位信息集中到主进程中，由主进程负责完成所有元素的最终排位。该并行算法描述如下：

算法5.4 枚举排序并行算法

输入：无序数组a[1]…a[n]

输出：有序数组b[1]…b[n]

Begin

(1)Po播送a[1]…a[n]给所有Pi

(2)**for all** Pi**where** 1≤i≤n**para-do**

(2.1)k=1

(2.2)**for**j= 1 **to**n **do**

if (a[i] > a[j]) **or**(a[i] =a[j] **and**i>j) **then**

k = k+1

end if

end for

(3)Po收集k并按序定位

End

2.3 归并排序

使用均匀划分技术，实现了 `p==3` 和 `p==4` 两种情况。

均匀划分技术

* 划分方法

  n个元素A[1..n]分成p组，每组A[(i-1)n/p+1..in/p]，i=1~p

* 示例：MIMD-SM模型上的PSRS排序

begin

(1)均匀划分：将n个元素A[1..n]均匀划分成p段，每个p_i处理A[(i-1)n/p+1..in/p]

(2)局部排序：p_i调用串行排序算法对A[(i-1)n/p+1..in/p]排序

(3)选取样本：p_i从其有序子序列A[(i-1)n/p+1..in/p]中选取p个样本元素

(4)样本排序：用一台处理器对p^2个样本元素进行串行排序

(5)选择主元：用一台处理器从排好序的样本序列中选取p-1个主元，并播送给其他p_i

(6)主元划分：p_i按主元将有序段A[(i-1)n/p+1..in/p]划分成p段

(7)全局交换：各处理器将其有序段按段号交换到对应的处理器中

(8)归并排序：各处理器对接收到的元素进行归并排序

end.

3 运行时间

30000个乱序数据，数据范围是[-50000, 50000]，每个排序程序运行100次，取平均值，结果如下：

	快速排序	枚举排序	归并排序
串行	5ms	2156ms	9ms
并行	1ms	357ms	4ms

程序原始输出为：

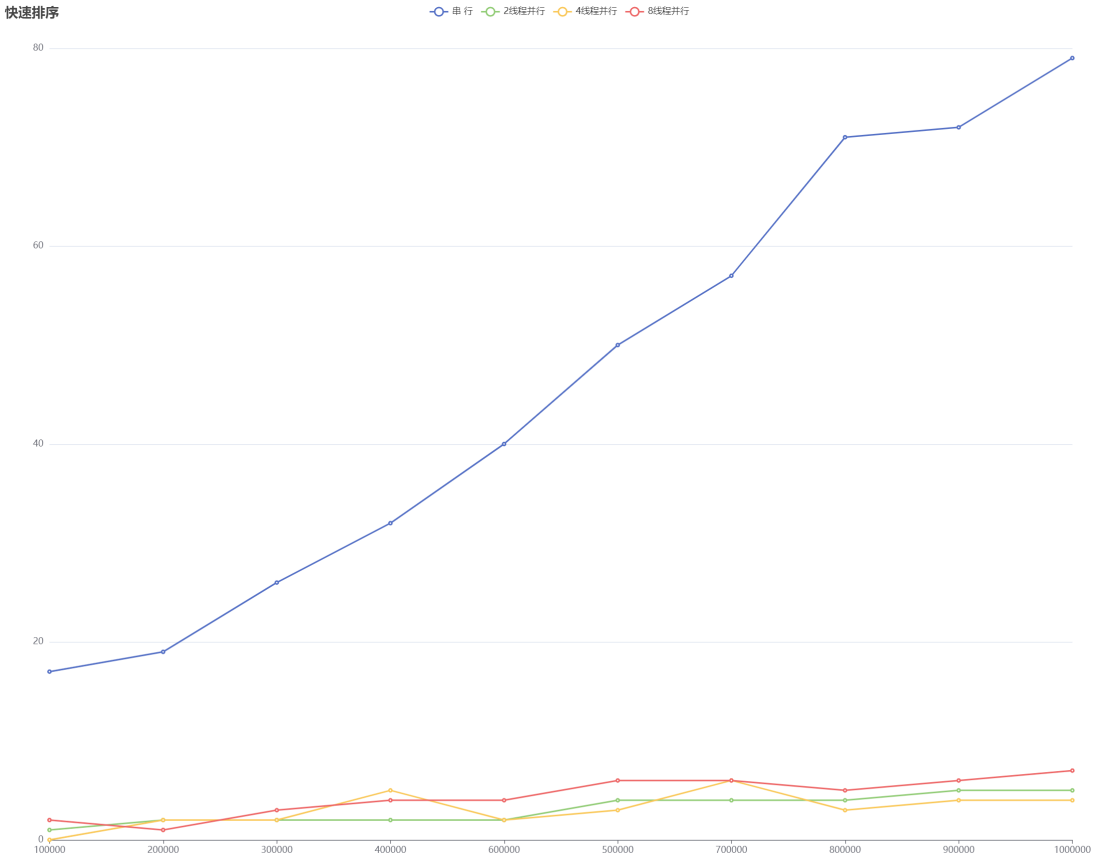
```
测试结果:true;  串行快排:5ms
测试结果:true;  2线程并行快排:1ms
测试结果:true;  4线程并行快排:0ms
测试结果:true;  8线程并行快排:1ms
==================================
测试结果:true;  串行枚举:2156ms
测试结果:true;  2线程并行枚举:1109ms
测试结果:true;  4线程并行枚举:607ms
测试结果:true;  8线程并行枚举:384ms
测试结果:true;  16线程并行枚举:357ms
测试结果:true;  32线程并行枚举:361ms
==================================
测试结果:true;  串行归并:9ms
测试结果:true;  3线程并行归并:4ms
测试结果:true;  4线程并行归并:8ms
```

4 结果分析

4.1 快速排序

发现30000个数据对于快速排序而言太少，体现不出并行算法与串行算法真正的效率比，故取10万到100万数据，，以数据集大小为横轴，排序时间为纵轴，绘成下图。

可以看到，串行算法执行时间和数据集大小具有线性关系，符合常理。但是三个并行程序皆变化不大，应该是效率较高，数据集过小导致；除此之外，随着数据集变大，有的并行算法，比如4线程并行，并不是单调递增，应该是由于程序实际是运行在虚拟机上，受操作系统调度所导致，各线程实际运行时间并不完全相同。



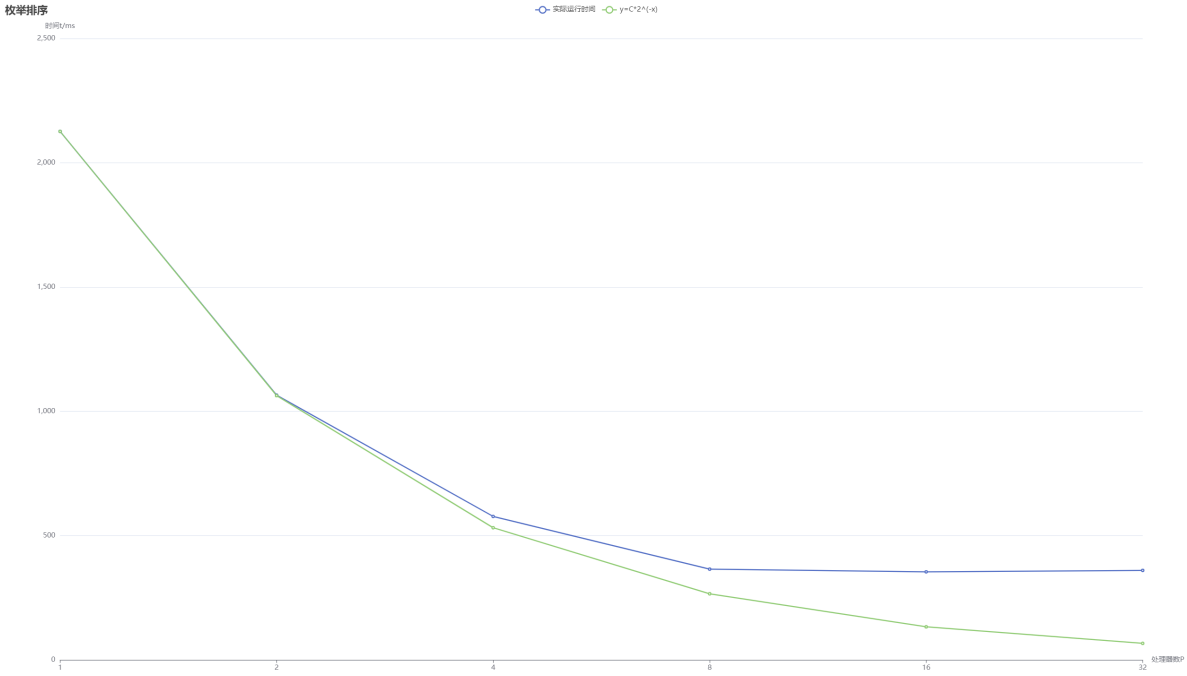
4.2 枚举排序

取不同并行数，运行结果如下：

```
测试结果:true;   串行枚举:2126ms
测试结果:true;   2线程并行枚举:1065ms
测试结果:true;   4线程并行枚举:577ms
测试结果:true;   8线程并行枚举:365ms
测试结果:true;   16线程并行枚举:354ms
测试结果:true;   32线程并行枚举:360ms
```

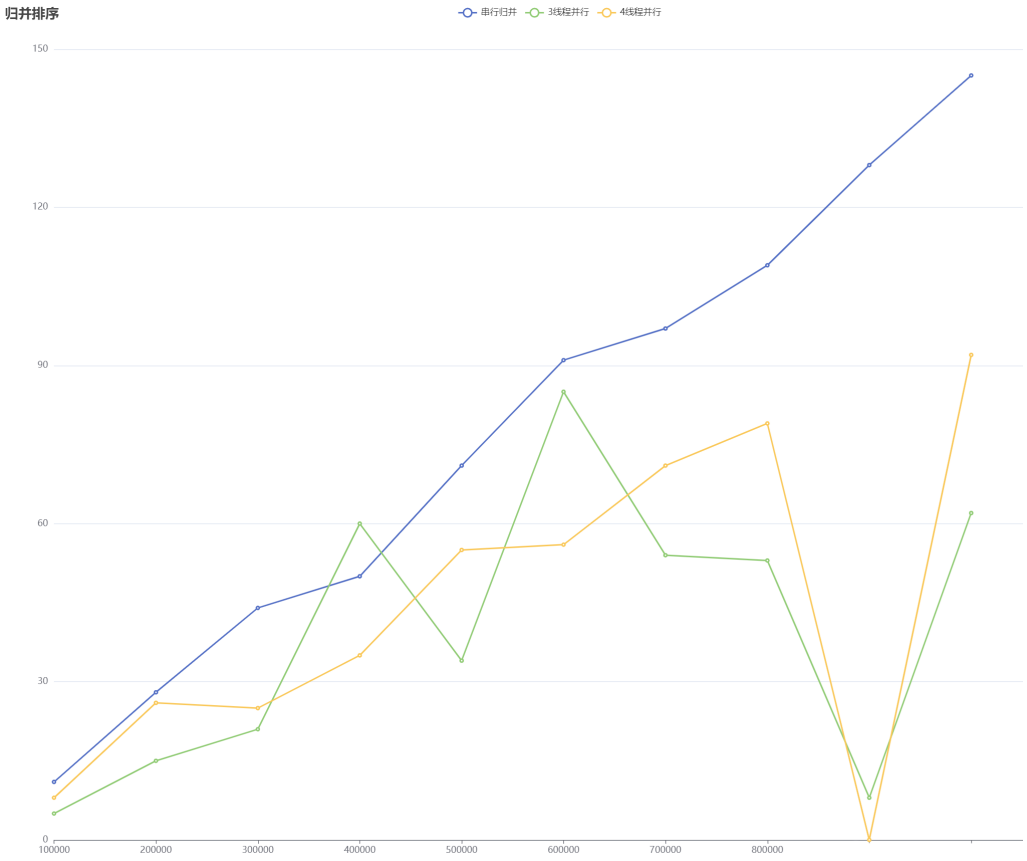
以横轴为处理器数`p`，纵轴为排序耗时`t`，绘图如下，绿色折线为$t = C * 2^p$，用于辅助分析。

发现当处理器个数翻倍，程序实际运行时间略高于原运行时间的一半，这符合常理，因为并行线程间有额外的通信开销等；当处理器个数越来越多，程序执行时间几乎不再下降，此时并行的额外开销已经大于并行算法节省的时间。



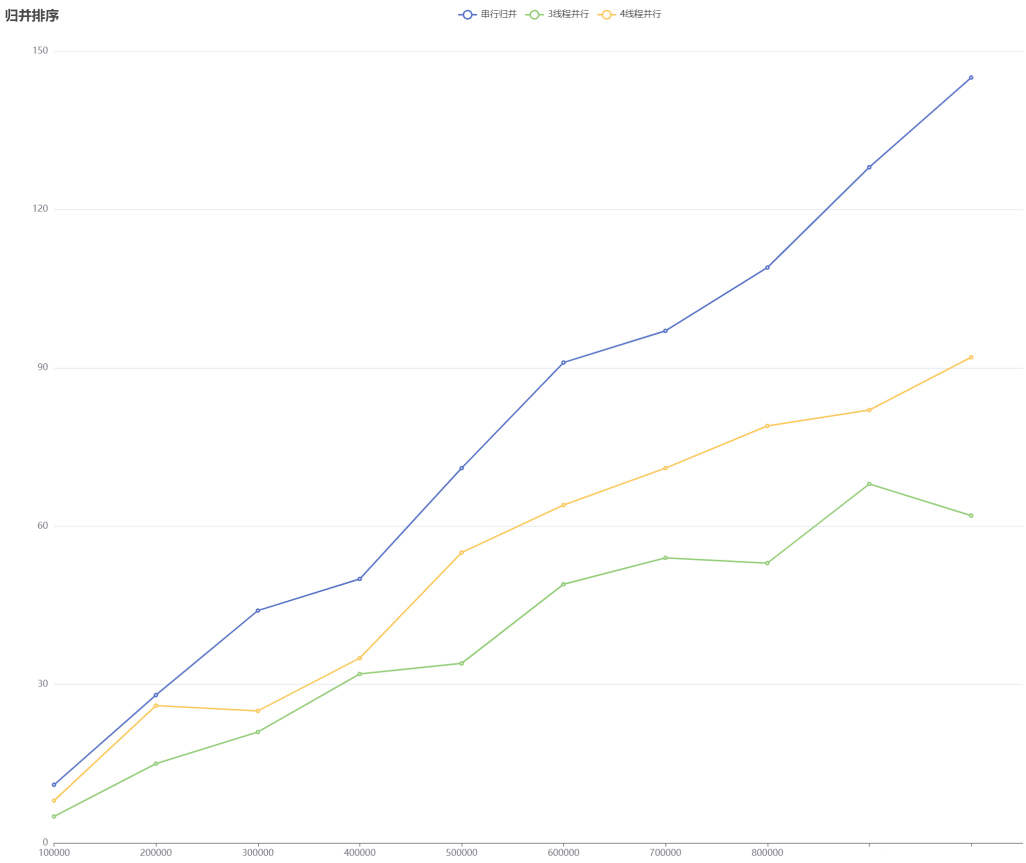
4.3 归并排序

发现30000个数据对于归并排序而言也太少，故也取10万到100万大小的数据集，以数据集大小为横轴，排序时间为纵轴，绘成下图。



发现排序时间杂乱无章，没有规律，考虑到操作系统会进行进程调度，排序时间具有随机性，受其他同一时间志在运行的进程影响。故多次运行排序程序，取运行时间均值，绘成下图。

发现排序时间3线程并行最快，串行最慢，而4线程并行却慢于三线程，应该是由于归并排序并行化的额外开销较大，3线程已满足需求，4线程的并行开销已经开始超过并行算法节约的时间。



5 技术要点

5.1 同步屏障

部分地方需要设置同步障，例如在均匀划分中，局部排序阶段需要等待所有处理器都完成排序后才可以进入下一步。

解决办法是使用`CountDownLatch`类，初始化参数为线程数`N`，每个线程的`run()`方法最后都会执行`countDown()`方法，在主进程接收到`N`个线程的结束信号之后才会继续执行。

```
... // other code
CountDownLatch end_signal = new CountDownLatch(8);
... // other code
end_signal.await();
... // other code
```

```
@Override
public void run() {
    ... // other code
    end_signal.countDown();
}
```

5.2 全局交换

借鉴了快速排序的`partition`方法，实现了本地的全局交换。

```
// 全局交换
int p0 = left - 1, p1 = left - 1;
for (int k = left; k <= right; k++) {
    if (nums[k] <= samples[3]) {
        p0++;
        p1++;
        swap(nums, k, p1);
        swap(nums, p0, p1);
    } else if ((nums[k] <= samples[6]) && (nums[k] > samples[3])) {
        p1++;
        swap(nums, k, p1);
    }
}
```

5.3 时间测量

因为现在操作系统都实现了时分复用，用户进程运行在虚拟机上，由内核进程统一调度，故一次运行结果不能反映算法实际效率，需要多次运行取平均值。

```
long time = 0;
for (int k = 0; k < rerun_times; k++) {
    load();
    long msort_start = System.currentTimeMillis();
    MergeSort.msort(nums, 0, nums_len - 1);
    long msort_end = System.currentTimeMillis();
    time += msort_end - msort_start;
}
System.out.println(time / rerun_times + "ms");
```

5.4 并行数选择

综合考虑算法效率与并行开销，发现快速排序`p==2`，枚举排序`p==8`，归并排序`p==3`时效果最好，既可提高排序效率，又不浪费处理器资源。