

并行程序设计大作业报告

范洪宇 软博 18 班 2018312541

一. 选题介绍

➤ 分段双调排序算法

分段双调排序算法是一种非常适合并行化的排序算法，算法即给出分成 m 段的 n 个浮点数，输入的数据按照段号有序，但是每个段内部是无序的。经过分段双调排序算法处理后，每个段中数据按照递增或者递减的顺序排序。在单线程情况下，算法的时间复杂度是 $O(n * \log n * \log n)$ ，并行处理后可以大大加速。

因为分段双调排序段间不影响的特点，非常适合并行化，常被并行化后用在网络拓扑排序中。除了并行化，大作业中还有一些创新点，除了输入数据申请空间进行存储，在排序的过程中算法并未申请任何额外的内存空间，仅借助已有的内存空间进行排序，从而能够保证空间安全。

➤ 双调排序算法

分段双调排序每个段之间没有数据关联，排序算法执行过程中，每个段都是使用的双调排序算法，下面会对双调排序算法进行一个简单的介绍，资料来源 (<https://blog.csdn.net/shanwenkang/article/details/82811130>):

双调序列

在了解双调排序算法之前，我们先来看看什么是双调序列。双调序列是一个先单调递增后单调递减（或者先单调递减后单调递增）的序列。

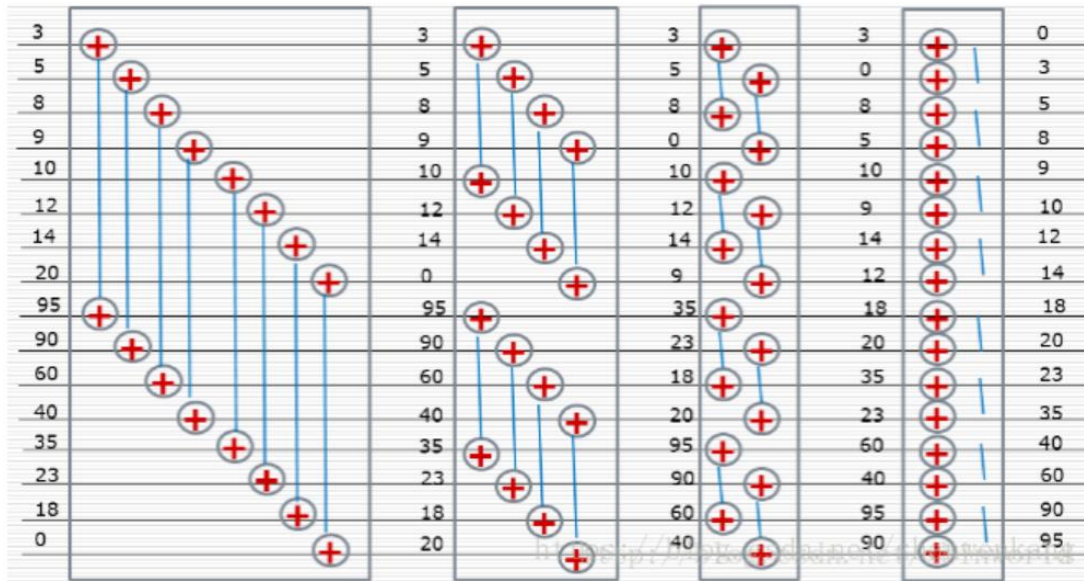
Batcher定理

将任意一个长为 $2n$ 的双调序列 A 分为等长的两半 X 和 Y ，将 X 中的元素与 Y 中的元素——按原序比较，即 $a[i]$ 与 $a[i+n]$ ($i < n$) 比较，将较大者放入 MAX 序列，较小者放入 MIN 序列。则得到的 MAX 和 MIN 序列仍然是双调序列，并且 MAX 序列中的任意一个元素不小于 MIN 序列中的任意一个元素[2]。

双调排序

假设我们有一个双调序列，则我们根据Batcher定理，将该序列划分成2个双调序列，然后继续对每个双调序列递归划分，得到更短的双调序列，直到得到的子序列长度为1为止。这时的输出序列按单调递增顺序排列。

见下图：升序排序，具体方法是，把一个序列 $(1...n)$ 对半分，假设 $n=2^k$ ，然后1和 $n/2+1$ 比较，小的放上，接下来2和 $n/2+2$ 比较，小的放上，以此类推；然后看成两个 $(n/2)$ 长度的序列，因为他们都是双调序列，所以可以重复上面的过程；总共重复 k 轮，即最后一轮已经是长度是2的序列比较了，就可得到最终的排序结果。



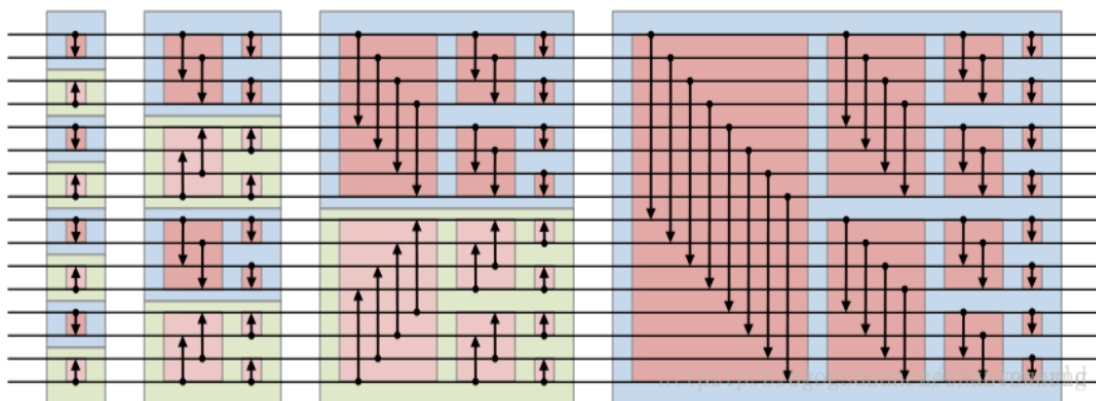
任意序列生成双调序列

前面讲了一个双调序列如何排序，那么任意序列如何变成一个双调序列呢？

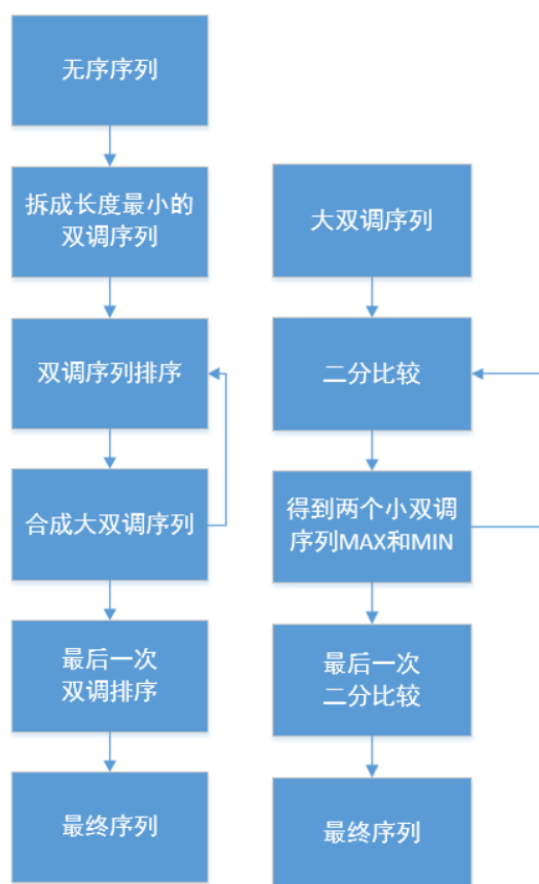
这个过程叫Bitonic merge, 实际上也是divide and conquer的思路。和前面sort的思路正相反，是一个bottom up的过程——将两个相邻的，单调性相反的单调序列看作一个双调序列，每次将这两个相邻的，单调性相反的单调序列merge生成一个新的双调序列，然后排序（同3、双调排序）。这样只要每次两个相邻长度为 n 的序列的单调性相反，就可以通过连接得到一个长度为 $2n$ 的双调序列，然后对这个 $2n$ 的序列进行一次双调排序变成有序，然后在把两个相邻的 $2n$ 序列合并（在排序的时候第一个升序，第二个降序）。 n 开始为1，每次翻倍，直到等于数组长度，最后就只需要再一遍单方向（单调性）排序了。

以16个元素的array为例，

1. 相邻两个元素合并形成8个单调性相反的单调序列，
2. 两两序列合并，形成4个双调序列，分别按相反单调性排序
3. 4个长度为4的相反单调性单调序列，相邻两个合并，生成两个长度为8的双调序列，分别排序
4. 2个长度为8的相反单调性单调序列，相邻两个合并，生成1个长度为16的双调序列，排序双调排序示意图



算法实现的核心流程如下左侧的流程图所示，双调序列排序每一次处理双调序列的过程如下图右图所示：



➤ 并行化选择

本作业选择使用 C++语言实现，使用的多线程库函数为 Pthread 库，为了最后代码检查简单，所有的代码实现在一个文件中，支持 g++的操作系统都可以编译运行。

二. 算法实现环境以及实现方案

➤ 运行平台及环境

实验测试的台式机环境是 Ubuntu 16.04 LTS，CPU 配置为 Inter-i7-7700，4 核八线程，16GB 内存，240GB SSD。

实验测试使用的服务器环境是 Ubuntu 16.04 LTS，CPU 配置为 Inter-E5-2620，12 核 24 线程，32GB 内存，2TB HD。

实验基本上是在命令行里面运行的，调试环境使用到了 VS code。实验中使用 g++编译即可，无需其他依赖。此外因为台式机和服务器上面配置不同，不同

机器上实验数据不具备可比性，我们只把同一台机器上获取的结果进行对比分析。

➤ 实验编译以及运行

所有代码都在 `segBitonicSort.cpp` 文件中，只需要在命令行运行

```
g++ segBitonicSort.cpp -o sort -lpthread
```

就可以生成可执行文件。

Windows 下执行 `sort.exe`

Linux 类系统下执行 `./sort`

即可，输入输出测例已经在代码内部写好了

➤ 实现方案

算法的原理和简单的例子已经在第一章介绍双调排序中给出了，这里不再重复介绍，仅介绍一些实现的技术细节和流程。

体现并行排序算法的提升需要一定规模的数据，所以首先是数据生成问题。算法中实现了生成数据的部分，生成的数据作为排序算法的输入。生成数据可以指定生成数据的个数以及分段的段数，比如设置生成 1 千万个数据，分成 10 个段。生成的数据会保存在 `data.txt` 文件中，其中包括 1 千万个数据以及每个数据所在的段的信息。

接下来 `data.txt` 会作为排序算法的输入，因为体现并行的优势，所以进行了单线程和多线程情况的对比，最后会生成三个输出文件，分别为 `output_single.txt`，`output_mt.txt` 以及 `result.txt`。`output_single.txt` 文件是在单线程情况下排序后的输出文件，`output_mt.txt` 文件中是在多线程线程下排序后的输出文件，这两个文件的内容是完全一样的。`result.txt` 中存放了两种不同情况下运行时间以及耗时对比。

程序的核心接口描述如下：

```
void segmentedBitonicSort(float* data, int* seg_id, int* seg_start, int n, int m);
```

输入数据中，`data` 包含需要分段排序的 n 个 `float` 值，`seg_id` 给出 `data` 中 n 个元素各自所在的段编号。`seg_start` 共有 $m+1$ 个元素，前 m 个分别给出 $0..m-1$ 共 m 个段的起始位置，`seg_start[m]` 保证等于 n 。

`seg_id` 中的元素保证单调不下降，即对任意的 $i < j$ ，`seg_id[i] ≤ seg_id[j]`。
`seg_id` 所有元素均在 0 到 $m-1$ 范围内。

输出结果覆盖 `data`，保证每一段内排序，但不改变段间元素的顺序

程序的核心并行单元为 `Void* bitonicSort(Void* para)`，其中封装了双调排序的算法实现，给定无序数组段的起始和结束位置，算法运行后能够给出排序后的数

组。大作业的创新点体现在，排序算法会利用输入数据的数组空间进行排序，在整个排序的过程中不会动态申请任何的内存空间，防止不安全的访问，能够在并行网络拓扑环境中限制空间申请。

算法中同时进行了单线程排序以及多线程排序，线程的数量可以提前设置，如果线程设置数量多于段的数量，则线程数量即为段的数量，在运行过程中保证线程数量不会超过 CPU 支持的线程数量。

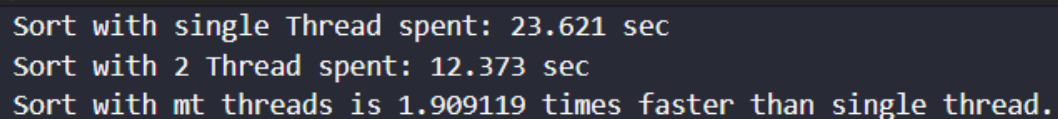
算法中还提供了格式化输出和时间监控的部分，能够将单线程和多线程情况下的排序结果输出到文件，并且对不同过程的时间进行记录 and 对比。

三. 实验结果对比分析

➤ 1 千万数据 2 数据段 && 单线程 VS 多线程

第一组对比试验的配置如下，在自己的台式机上面，随机生成 1 千万个数值在 0-11234567 之间的浮点数，并且随机分成 2 个数据段，然后在单线程与多线程的情况下进行实验。

已知在单线程情况下，算法的时间复杂度为 $O(n * \log n * \log n)$ ，1 千万个数据分成 2 段。假设 CPU 处理 1 千万次的时间约为 1s，那么估算可得排序的时间约为 40s，经过试验对比，得到如下结果：



```
Sort with single Thread spent: 23.621 sec
Sort with 2 Thread spent: 12.373 sec
Sort with mt_threads is 1.909119 times faster than single thread.
```

由图片可得，单线程情况下处理 1 千万个数据，2 个数据段的排序花费了 23.6s 左右的时间，多线程并行情况下花费了 12.3s 左右的时间，基本在 1.9 倍的提升。分析结果，CPU 性能越来越好，单线程情况下 CPU 每 1s 可以处理的数据量应该超过 1 千万次，所以单线程情况下为 23.6s，而不是 40s 也是合理的实验结果，时间在数量级上满足逻辑即可。此外，数据量占用量并不大，CPU 和内存比较宽裕，能够比较靠近理论上的提升。

➤ 3 千万数据 6 数据段 && 单线程 VS 六线程

第二组对比试验的配置如下，在自己的台式机上面，随机生成 3 千万个数值在 0-11234567 之间的浮点数，并且随机分成 6 个数据段，因为 CPU 是 4 核八线程，在单线程与六线程的情况下进行实验。

和第一组对比试验一样，检测测试了一下 CPU 处理 1 千万次数据所花费的时间，之前估算在 1s 左右，经过测试，只需要 0.6s 左右，因为之前衡量算法的时候，基本上默认是 1 千万次运算 CPU 时间约为 1s，不过已经是几年前了，CPU 处理的性能变得更强了，所以耗费的时间更少了，是合理的。所以根据双调排序的时间复杂度，可以估算出理论上单线程，3 千万数据分成 6 个数据段下排序的时间约为 80.1s。经过实验对比，得到如下的结果：

```
Sort with single Thread spent: 71.748 sec
Sort with 6 Thread spent: 17.799 sec
Sort with mt_threads is 4.030927 times faster than single thread.
```

由图片可得，在单线程情况下，处理 3 千万数据，6 个数据段需要消耗 71.7s，但是在六个线程情况下需要消耗 17.8s，速度提升了 4.03 倍。速度的提升比起理论上 6 倍速度还是有一定的差距，因为 CPU 竞争，内存访问，文件读写等都可能存在等待和调度，并且不同数据排序本身耗时不同，时间复杂度仅代表一个普遍情况。综上所述，时间是合理的。

➤ 1 亿数据 10 数据段 && 单线程 VS 十线程

第三组对比试验的配置如下，第三组实验是在服务器上跑的，随机生成 1 亿个数值在 0-11234567 之间的浮点数，并且随机分成 10 个数据段，然后在单线程与十线程的情况下进行实验。

首先估算一下时间复杂度，CPU 处理 1 千万次运算时间约为 0.6s。目前的配置是 1 亿个数据，分成 10 个数据段，那么时间复杂度可以计算为 $10 * 10^7 * \log 10^7 * \log 10^7 * 10^{-7} * 0.6$ ，经过计算可得约为 294s。那么经过对比试验，可以得到如下的结果：

```
Sort with single Thread spent: 292.389 sec
Sort with 6 Thread spent: 35.173 sec
Sort with mt_threads is 8.312839 times faster than single thread.
```

由图可得，1 亿个数据分成 10 个数据段，在单线程情况下排序时间为 292.4s，跟理论上排序时间基本相同，在十线程并行情况下排序花费了 35.1s 的时间，加速比为 8.31 倍。分析结果，与预期结果基本相符，提升非常明显，但是无法达到理论上 10 倍的速度，因为数据段之间情况不同，先结束的线程需要等待后续线程，并且对于存储，内存等访问也有竞争和调度，有一定的影响。总而言之，实

验结果是符合预期的，也体现出了分段双调排序并行的可行性与高效性。

➤ 3 亿数据 20 数据段 && 单线程 VS 二十线程

第四组对比试验的配置如下，第四组实验是在服务器上跑的，随机生成 3 亿个数值在 0-11234567 之间的浮点数，并且随机分成 20 个数据段，然后在单线程与二十线程的情况下进行实验。

首先估算一下时间复杂度，目前的配置是 3 亿数据，分成 2 个数据段，在单线程情况下排序的理论时间计算约为 933s。经过对比试验，得到结果为：

```
Sort with single Thread spent: 900.183 sec  
Sort with 6 Thread spent: 58.237 sec  
Sort with mt_threads is 15.305812 times faster than single thread.
```

由上图可得，在单线程情况下，排序时间为 900.2s，符合预期，在二十个线程的情况下排序花费的时间为 58.3s，加速比约为 15 倍。实验结果基本符合预期的目标。

➤ 10 亿数据 50 数据段 && 单线程 VS 二十二线程

第五组对比试验的配置如下，第五组实验是在服务器上跑的，随机生成 10 亿个数值在 0-11234567 之间的浮点数，并且随机分成 50 个数据段，然后在单线程与二十线程的情况下进行实验。相比于前四组实验，这组实验比较特殊，因为首先数据量非常大，而且服务器支持的核心数是小于段落数的，所以即使多线程，后续段落也需要等待才能分配到资源进行排序。本组实验作为前几组实验的对比试验进行观察与总结。

因为数据段的数量多于线程数量，所以并行加速比应该无法按照原始的情况进行衡量。单线程情况下，处理 10 亿数据，50 个数据段，经过估算的理论时间约为 3197s。经过对比试验之后，实验结果为：

```
Sort with single Thread spent: 3163.315 sec  
Sort with 2 Thread spent: 129.802 sec  
Sort with mt_threads is 24.370364 times faster than single thread.
```

由上图可得，10 亿数据 50 个数据段在单线程情况下进行排序所花费的时间为 3613.3s，在 22 个线程情况下花费的时间是 129.8s。加速比为 24.37 倍。因为数据段数量是多于允许使用的线程的数量的，所以并行化一开始运行式有很多数据段是需要等待的，等待前面线程释放之后再继续进行排序。所以加速的比例肯定是

无法到达接近 50 倍的。目前的结果应该也在合理的范围之内。

➤ 注意事项

因为每次运行都是动态的随机生成数据文件，1 亿个数据文件大小就已经超过 1GB 了。所以提交的时候不提交生成的数据里，最后提交的代码里面保存了 1 千万数据，两个数据段。单线程和双线程的对比试验，如果运行测试的话，大约半分钟能够运行完。其余的可以自行修改主函数里面的配置参数来进行修改。

四. 总结和感想

经过对分段双调排序的实现以及并行化处理，还是学到了并行程序设计实现方面的一些知识和方法，并且自己动手收获很多。

首先是双调排序的时间复杂度为 $O(n * \log n * \log n)$ ，算不上是性能最优的排序算法，但是其先天的可并行化处理的优点使其经常被使用在并行环境之下，综合性能良好。其次的感受就是使用多个线程，受限于线程调度，资源竞争，数据传输等影响，提升的效率是要小于预期的，这也是可以预见的，而且实验结果也是比较合理的，满足预期。

大作业实现过程中还是遇到了很多的问题，最难的在于算法的实现，其次就是基于创新点，不动态申请任何内存空间，需要对现有的数据空间进行划分，在排序过程中不断地覆盖和重复利用，保证空间访问的安全性。在这一点实现上调试花费了比较多的时间。

最后感谢向东老师的授课，以及助教的辛勤付出，通过《并行程序设计》这门课，复习了很多之前学习过的算法，排序，矩阵运算，图论中的一些方法等，而且对于并行程序实现进行了一些了解和动手实践，收获很多。