外部排序技术报告

王凯菲[1]

（1. 西安电子科技大学 计算机科学与技术学院，陕西 西安 710126）

摘要

排序问题常见于各项工程中，在内存空间不足的情况下，就需要借助外存来进行外部排序。现阶段广泛使用的方法是先将待排序数据分割成内存可容纳的部分，将其内排后写入外存，随后对生成的各内部有序序列进行归并排序并得到最终结果。外部排序问题一般数据量极大，所以对其执行效率的提高也就尤为重要。

本文针对 512M 内存限制下的 2.5 亿不规则浮点数排序问题进行研究。实现了外部排序算法，重点展示了浮点数编码技术，对两种多线程实现方案进行了讨论。这些技术的实现大大提高了程序整体的执行效率。

选取 4.28GB 数据集，在 2.30GHz 四核 CPU，8GB 内存的硬件条件下测试该程序，最快 77s 即可完成全部任务。最后，本文还讨论了一些潜在的优化手段。

关键词：外部排序、败者树、编解码、多线程

1. 问题简介

排序是指将一组无序元素按照某种特定规则整理成有序序列的过程。在日常的数据处理中，一般认为有 1/4 的时间用在排序上，而对于安装程序，多达 50%的时间花费在对表的排序上。因为应用环境千差万别，也就诞生了各种不同的排序方法。

排序方法分为两大类：内部排序和外部排序。内部排序是指在排序期间数据元素全部存放在内存的排序。外部排序是指排序期间元素个数太多不能同时存放在内存，必须根据排序过程的要求，不断在内外存之间移动的排序。

外部排序主要分为两个阶段：第一阶段建立为外排序所用的内存缓冲区。将输入文件划分为若干段，用某种有效的内部排序方法对各段进行排序，然后再将排序后的内容写入外存。这些经过排序的段叫做初始归并段。第二阶段把第一阶段生成的初始归并段加以归并，一趟趟地扩大归并段和减少归并段个数，直到最后归并成一个大归并段（有序文件）为止。

在本文中，主要解决了对 2.5 亿个十进制浮点数从小到大排序且内存使用峰值不超过 512M 的问题。本作业的主要难点在于浮点数编码和多线程实现上，之后将对其和程序主要实现过程进行详细阐述。

2. 程序设计与实现

2.1 划分归并段

2.1.1 浮点数编码

浮点数编码是本作业的一个难点。输入文本虽满足 IEEE754 双精度浮点数标准，但因其格式不确定，且有非法输入条目，所以不能直接读入比较，需要用编解码技术进行操作。本文采取了以下编解码方案。

2.1.1.1 初步思想

对于 IEEE754 双精度浮点数标准，一个数字由尾数和幂数来表示。且尾数不会大于幂数的底数。所以对于十进制浮点数，合法输入不会出现类似于 13.2 即尾数大于 10 的情况。

分析可得，指数部分优先级要大于小数部分。当指数不相同时才有必要进行小数部分的比较。所以我们可以用 64 位整型来对输入条目进行编码。将指数部分放在前而小数部分在后。当指数、小数部分符号不同时，其绝对值大小与最终结果之间有着不同的影响。具体如表 1 所示。

区间	指数绝对值	小数绝对值
小数为正指数为正	正相关	正相关
小数为正指数为负	负相关	正相关
小数为负指数为正	负相关	负相关
小数为负指数为负	正相关	负相关

表 1 小数指数绝对值大小在不同符号条件下与整体大小的关系

根据上表可采取这样的编码方案：将每一合法条目编码为十进制整数。第一位十进制数为指数符号位，第二到四位为指数绝对值，第五到十四位为小数绝对值。条目的符号即为编码整数的符号。

当指数为负时，编码指数绝对值需要用最大指数值 1000 减去当前指数绝对值，以满足表 1 中的相关性要求。当指数为正时第一位为 2，指数为负时第一位为 1，以保证指数符号不同时的正确性。

如将 9.123E+12 编码成：

2 012 9123000000

指数符号位 指数绝对值 小数绝对值

将-5.43234E-100 编码成：

\- 1 100 5432340000

小数符号 指数符号位 指数绝对值 小数绝对值

进行编码操作后对每个条目可直接比较，加快了程序运行速度，降低了比较操作的逻辑复杂度。

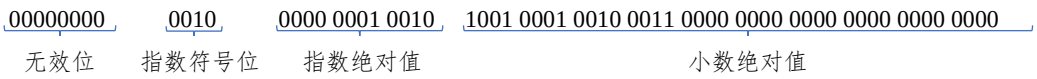
2.1.1.2 位运算与 BCD 编码[1]

上节的方案需多次用到乘法操作，而整型乘法耗时较多。可将每一位十进制数字编码成四位二进制，即 BCD 编码来避免乘法操作。而对应于上节中负指数编码需用 1000 减去当

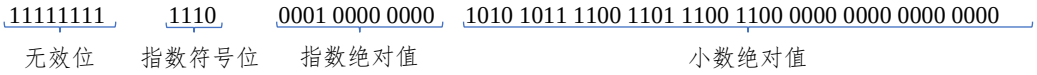
[1] .此节十分感谢永卿师兄的帮助。

前指数绝对值的操作，直接对指数绝对值取反即可达到相同效果。用 64 位整型存储结果，10 个小数数字，3 个指数数字，1 个指数符号，每个用 4 位二进制进行存储，共需 56 位。剩余 8 位，正数全为 0，负数全为 1，为无效位。

如将 9.123E+12 编码成：



将-5.43234E-100 编码成：



在进行 BCD 编码时，可直接用位操作给整型赋值，以加快编码速度。如要将第 3 个四位二进制（指数符号位）赋值成 2，则将编码数字和 0x0020000000000000 相与即可（需注意各位赋值时相互影响）。

2.1.1.3 解码部分

通过编码操作，我们已经将问题简化为 64 位整型的排序。在写出最终结果时，需要将编码后的整型转化为合法的字符串格式。简单地说，解码过程就是编码的反操作，提取出每个数字并填入字符串的正确位置即可。如要想提取-5.43234E-100 的第一位指数，只需进行以下操作：

1. 将编码后整型取相反数。
2. 把上步结果取反。
3. 将上步结果与 0x000F000000000000 做与运算。

2.1.2 划分归并段

因为读写磁盘包含机械动作，所以磁盘读写时间远大于内存运算时间。为了减少磁盘读写次数，提高运行速度，可以减少初始归并段 m 的个数。

在总记录数一定时，要想减少归并段个数，就必须增大归并段长度。若规定每个归并段等长，则此长度应该根据它的内存工作区空间大小而定，因而 m 的减少也就受到了限制。可以用败者树来突破这个限制：在占用同样内存的情况下，可以生成平均比原来等长情况下大一倍的初始归并段，从而减少参加多路归并的归并段个数，减少 IO 次数。

2.1.2.1 败者树介绍

败者树是一种类似于完全二叉树的数据结构。在败者树中，每个叶结点存放待比较值，非叶结点记录其两个孩子结点中败者的序号。若有 k 个叶子结点，则设置 k 个非叶子结点，其中第 0 号非叶子结点存储最终胜者的序号。我们将胜者定义为“最小”来引入下面的例子。若有一颗叶子数为 5 的败者树，叶结点值分别为｛17，05，10，29，1｝，则其构建的败者树如图 1 所示。

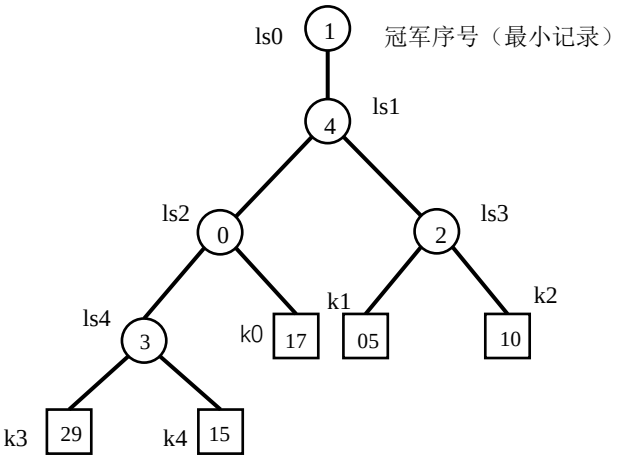


图 1 败者树形态介绍

当前冠军(即最小值)输出后，在 k1 中读入下一个值如 44，则图 1 中的败者树将做出如下调整：ls3 调整为 1、ls0 调整为 2。即下一个冠军为 2 号叶子结点，其值为 10。调整后的败者树形态如图 2。

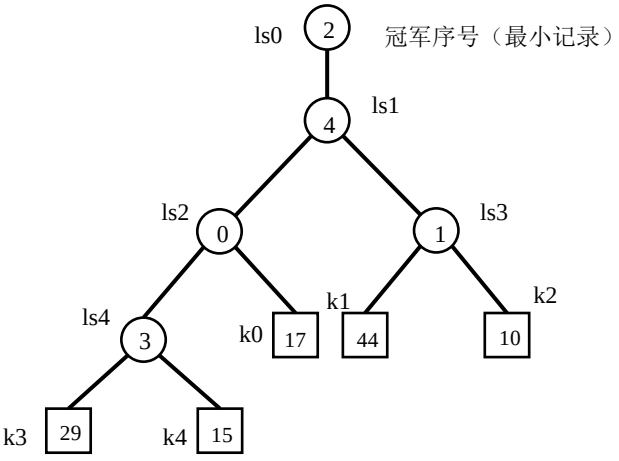


图 2 败者树调整示例

2.1.2.2 划分归并段算法

败者树中的每个叶子结点都由段号、待比较值所标识。胜者这样定义：段号小的或段号相同但值更小的。算法具体步骤如下：

1. 从输入文件中把 k 个记录读入败者树的 k 个叶子结点中，并初始化败者树。
2. 利用败者树选出最优值（设其序号为 q），将其值存入 LastKey 中。
3. 将此最优值写入当前输出文件中。
4. 若文件未读完，则从输入文件中读入下一个记录，用其值置换第 q 个叶子结点。若新写入的值比 LastKey 小，则将其段号赋值为 LastKey 的段号加一；否则赋值为 LastKey 的段号。
5. 重复 2~4 直到所选出最优值的段号与 LastKey 段号不等，此时当前初始归并段生成完毕，创建新文件以存放下一归并段。
6. 重复 2~5 重新开始选择和置换，直到输入文件全部读完。

2.2 多路归并

完成划分归并段后会得到 n 段段内有序待归并序列。在简单的二路归并中，每次从两个有序归并段提取各自最优元素进行比较，再将较优者写入生成段。重复这个步骤，最终将生成包含两个归并段全部元素的有序序列。

在 k 路归并时，也可以采用遍历 k 个归并段当前最优元素的方法来选取最优值。为了减少比较次数可以用败者树来进行选择。利用败者树，在 k 个记录中选取最优者，只需要$O(logk)$次比较。

进行 k 路归并的步骤如下：

1. 将 k 个内部有序的归并段分别作为 k 个叶子结点的输入。
2. 从败者树中选取当前最优值存入输出缓冲区。
3. 从相应输入缓冲区中读入一个新值到上一个最优值的叶结点位置。
4. 重复步骤 2、3，当某个输入缓冲区为空时，一律为需要其填入的位置填入 max 值。
5. 当选出的最优值为 max 时，结束循环。

2.3 多线程实现

由于内外存信息传输与 CPU 运行相比要慢得多，所以经常需要等待缓冲区填满或写出。为了改善这种状态，可以采用多线程技术来提高 CPU 的利用率。

2.3.1 简单开启多个线程进行相同的操作

本文对划分归并段部分开启了多个相同的线程。每个线程同时进行读入、编码、生成归并段、写出文件 4 个步骤。考虑到输入缓冲区容量有限，读入的文本不一定能全部编码而需要重新移动文件指针的情况，程序对读入文本和编码部分加互斥锁。整体并行性较高，CPU 使用率可达 95%左右。

2.3.2 类生产者消费者模式

多路归并阶段写出的是同一结果文件，无法并行。输入文件较多，每个文件都需要开设输入缓冲区,若按上节方案,内存代价太大。所以在该阶段本文采取类生产者消费者的模式，对于 k 路归并，设置 2k 个输入缓冲区和 2 个输出缓冲区即可较好地实现并行。具体实现思路为：

1. 假设给每一归并段固定分配了 2 个输入缓冲区，做二路归并。
2. 将各归并段内容读入各输入缓冲区 1 中。
3. 对输入缓冲区 1 中内容进行归并操作，写入输出缓冲区 1。
4. 归并的同时再从各归并段中读入内容到各输入缓冲区 2。
5. 当输出缓冲区 1 满后写入输出文件。
6. 进行第 5 步的同时，对输入缓冲区 2 归并，读取归并段内容到输入缓冲区 1。然后重复第 3~6 步直到输入文件读完。

这样在理想情况下，除最初的 k 个输入块的读入和最后一个输出块的写出外，其他所有输入、输出和内部归并都是并行执行的。该阶段 CPU 使用率为 67%左右。

3. 测试结果

测试在 2.5 亿个浮点数数据集（约 4.28GB）上进行，程序运行于 Windows10（64 位）操作系统，硬件条件为 Intel Core i5-6300HQ @ 2.30GHz 四核 CPU，8GB 内存。

划分归并段部分败者树大小设置为 1200000，多路归并部分败者树大小设置为 256。划分归并段部分输入文本大小和输入缓冲区大小随线程数而定。本文分别对划分归并段部分进行 1~8 线程测试，而多路归并部分由于其不能笼统的按某线程数测试，将该部分一直按 2.3.2 节中的多线程方式实现或不开启多线程。规定划分归并段、多路归并两个阶段都为单线程的情况为 0 线程。测试时各线程情况下分别运行 4 次，取其最小值为最终结果。各自耗时情况见下表。

线程数	0	1	2	3	4	5	6	7	8
耗时/s	155	129	89	81	79	78	78	77	78
输入缓冲区大小/MB	52	52	24.5	15	10	7	5.2	4	3
CPU 使用率 1	35%	35%	65%	90%	96%	97%	95%	97%	98%
CPU 使用率 2	29%	67%	68%	70%	68%	69%	68%	69%	70%

表 2 各线程数测试结果

表 2 中 CPU 使用率 1 为划分归并段部分 CPU 平均使用率，CPU 使用率 2 为多路归并部分 CPU 平均使用率，输入缓冲区大小表示划分归并段部分输入缓冲区大小设置。

由于内存限制，输入缓冲区大小与线程数成反比。在未采用多线程技术时，最快为 155s。采用多线程技术后，最快可达到 77s。开启多路归并部分并行及 2 线程划分归并段时，增速最为明显，耗时降到原来的 57%左右。

在开启大于 3 的线程数后，增速并不明显。首先因为在 3 线程时 CPU 使用率已经达到了 90%，继续增加线程，CPU 使用率并不会有明显的提升。其次因为内存限制，继续增加线程势必会减少输入缓冲区的大小，带来 IO 操作的增加。

4. 讨论

在划分归并段部分多线程技术的实现中，2.3 节所采用的方案不能对编码部分并行。但若能保证每次读入的字符串全部被编码，编码部分则可以安全并行，且避免了“移动文件指针”这一耗时较大的操作。一种可行的方案是用小的字符串（设容量为 x）来读入字符，而使用较大的输入缓冲区（设容量为 y）存编码后的内容。若输入缓冲区数量一旦大于 y-x 时，则停止读入，进行排序。用较小的内存浪费就可保证每次读入的字符串全部被编码。将这一思路实现后，阶段内磁盘读写速度差距极大，初步分析是由于 x 不能设置太大所致。在整体速度方面并没有明显提高。

接下来对划分归并段部分败者树大小设置进行讨论。该值设置越大，产生的归并段越少，多路归并就会越快。但同时树高越大，比较次数越多，且相应的输入缓冲区容量会减少，使得内外存交换数据次数增大。所以该值选取需要综合考虑。本文选取其为 1200000，对于 2.5 亿数据量，大约产生 108 个初始归并段。

对于本题目的 512M 内存限制，采用快速排序等内排算法（经过读入、编码、排序的步骤），并将输入缓冲区设置很大则可以产生更长的初始归并段。若内存限制更严格，使用败者树生成初始归并段的优势会更加明显。

参考文献

[1] 殷人昆，《数据结构：用面向对象方法与C++语言描述》，清华大学出版社，2007.6

[2] 马伟，《编写高质量代码：改善C程序代码的125个建议》，机械工业出版社，2016.1