针对海量浮点数的外排序算法

一．摘要

对于海量浮点数排序，由于计算机的内存空间有限，不能一次性地读入内存，故不能使用传统的快速排序等内部排序算法。为解决该问题，我们可以使用外排序。外排序将处理的数据放在读写较慢的外存储器上，通常采用“内排序-归并”的策略。在内排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件。最后在归并阶段将这些临时文件组合为一个大的有序文件，也即排序结果。

本文的外部排序算法在分块读写文件过程中加入了多线程。使用基数排序对小文件的浮点数进行排序，其速度明显优于快速排序。而在归并阶段，使用败者树进行比较，将时间复杂度由 O(n)降到了 O(logn)。为进一步提高排序速度，手动实现了字符串与浮点数互相转化的函数。加入多种优化后的外部排序算法的速度得到明显提高。

二．问题简介

1.对海量浮点数进行排序。输入文本由一般的十进制浮点数（符合 IEEE754 双精度浮点数标准）构成，每两个浮点数之间以换行符（\n）间隔，间隔符只有一个。另有非法输入条目。所谓“条目”只由若干字符组成的一行，如单词、英文标点符号等，他们中间可能混有一些数字，但条目本身不代表一个可计算的有理数。不会有很复杂的非法输入条目。例如输入文本 input.txt 内容如下：

```
3.14159268
6.62e-134
6.12345678E-11
4.9999999995
0.618
-1.01234567891234567
Hello
&1243#%$
```

2.输出文本是对输入文本排序后的结果，为普通的文本格式，每个结果独占一行。格式为：±*.*********E±***，一个*表示一个数字，共 10 个有效数字，不足补 0，多出的四舍五入。如 49.9999999995 输出 5.000000000E+001。注意，负数要有负号，正数就不必在前面写正号了，E 后面必须有正号或负号，E 是大写，不是小写。非法条目可以删除（不必输出到文本），但必须报告非法条目（可以直接显示在控制台上，包括每条的文本，以及全部非法条目的总数）。理论上来说，不管谁实现的程序，对于相同的输入文本，输出的文本应该是一模一样的。例如以上的 input.txt 的输出结果应该是：

```
-1.012345679E+000
6.620000000E-134
```

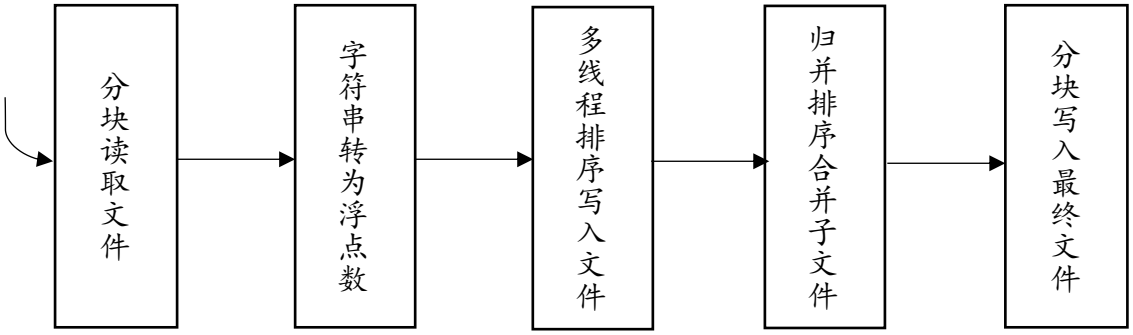
6.123456780E-011
6.180000000E-001
3.141592680E+000
5.000000000E+000

在对比时，可以使用 UltraEdit 或其他文本编辑器的文本对比功能。该类功能可以非常方便地发现两个文本不一致的地方，并高亮标示。我们一般使用 UltraEdit11a。

3.程序在运行(不管是单线程还是多线程)时,使用的内存峰值不可超过512M。建议优化下自己程序内部的参数，既可以充分利用这 512M 内存，同时又不能超过。鼓励实现多线程版本，但不强制要求。在统一测试时，设置单线程还是多线程自己决定，我们只对比绝对运行时间。在报告中要对内存利用以及线程使用进行讨论。

三．程序设计与实现

1.程序设计框架如下：



2.程序实现

2.1 分块读取文件

读取文件操作最终使用了 fstream 类中的 read 函数。起初，使用了多种读取文本的方法对一百万个浮点数进行读入写出，经对比发现，read 函数的速度快于 fscanf，后者的速度快于析取器。

在分块读取文件时，会出现读出的数据不完整的情况，由于浮点数之间是以换行符间隔的，所以，每次查找到最后一个读出的换行符，并将文件指针置于当前换行符的下一个字符，以便下次操作。

读取文件时使用了单线程，此时读文件操作相当于生产者。

2.2 字符串转化为浮点数

首要处理的是非法输入。假设非法输入中一定含有非法字符，即非数字、‘.’、‘+’、‘-’或‘\n’。对此，当每次读出的字符不是上述字符中的一个，表示当前为非法输入。于是，分别向前向后找到第一个‘\n’，则中间的所有字符即为一个非法条目，将其存入一个字符串中并输出。

在将字符串转化为浮点数的过程中，我先使用的是库函数 atof，但发现精度有问题，且速度又过慢，于是自己实现了一个 atof 函数，解决了上述问题，且

优化效果十分明显。输入并处理 2.5 亿个浮点数，使用 Microsoft Visual Studio 的 Debug 版本编译运行时，库函数需要 700S，优化后需要 400S。

2.3 多线程排序写入文件

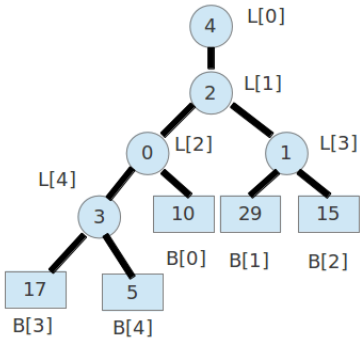
使用基数排序对每个小文件中的浮点数进行排序。基数排序、快速排序与库函数 sort 排序的速度差异很明显。在对一百万个浮点数进行测试时，发现使用未优化的快速排序速度为 10S，将快速排序进行四步优化后速度提升不明显，与库函数 sort 排序速度相仿，最终选择基数排序。一个存储在计算机中的 double 类型的浮点数，占据 64 位。从 0 到 63 位对数字的大小的重要性依次增大，因此可以像整数一样使用基数排序。但最终得到的结果为：正数从小到大排列，负数从大到小排列，解决方法有两种：第一，记录负数的个数，将排列之后的正负数翻转到自己的位置；第二，将正数的符号位由 0 置成 1，将负数的每一位取反，排序结束后在恢复。以上两种方法均可得到从小到大的浮点数序列，但经测试，发现，后一种方法用时更长，于是采用第一种方法。

写入文件操作，使用 fstream 类的 write 函数，速度优于 fprintf 和插入器。

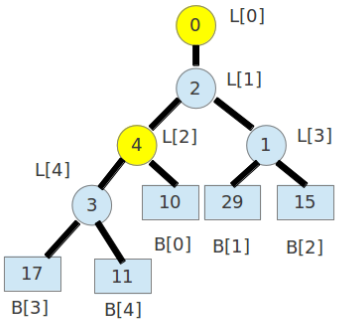
将写文件看作生产者，字符串转化为浮点数、基数排序、写入文件三个操作看作消费者，加入了单生产者-多消费者线程，速度得到提高。

2.4 归并排序合并子文件

归并排序的步骤是：每次从 K 个文件中的首元素中选一个最小的数，加入到缓冲区中，这样每次都要比较 K-1 次，因此算法的复杂度为 O((N-1)*(K-1))，而使用败者树进行比较，每次取出最小数只需比较 O(logK)次，故可以在 O((N-1)*logK)的复杂度下完成整个排序。构建败者树的过程如下图（1）所示：



图（1）



图（2）

因为要按从小到大排序，所以在每次比较中，小的为胜，大的为败。数组 B[0..4]存储从顺串中读入的数，L[0]存储最终的胜者（最小值）的位置，L[1..4]存储中间各比赛败者的位置。当前最小值为 5（B[4]），将 5 输出后，若新读入的数据为 11，则先与该组之前的败者 B[3]比较，胜后再与 B[0]比较，结果为败，则将下标 4 记录于 L[2]处，令胜者 B[0]继续向上与 B[2]比较，胜出后将下标记录到 L[0]，经过 3 次比较后得出新的最小值为 10(B[0])，如图（2）所示。

关于败者树的构建和每次读入新值后的调整步骤见下面代码：

```
//功能：创建败者树
//参数：run_t 结构体，  有序文件的个数
//返回值：空
```

```
void createLoserTree(run_t **runs, int n)
{
    for (int i = 0; i < n; i++)
    {
        ls[i] = -1;
    }
    for (int i = n-1; i >= 0; i--)
    {
        adjust(runs, n, i);
    }
}
//功能: 构建败者树, 每次读入新值调整败者树
//参数:run_t 结构体, 有序文件个数, 当前文件下标
//返回值: 空
void adjust(run_t **runs, int n, int s)
{
    int t = (s + n) / 2;
    int tmp;

    while (t != 0)
    {
        if (s == -1)
            break;
        if (ls[t] == -1 || runs[s]->buf[runs[s]->idx] >
            runs[ls[t]]->buf[runs[ls[t]]->idx])
        {
            tmp = s;
            s = ls[t];
            ls[t] = tmp;
        }
        t /= 2;
    }
    ls[0] = s;
}
```

2.5 分块写入最终文件

在写入最终文件前,需要将浮点数转化为字符串,然后存储在临时缓冲区中,待缓冲区满时,将所有字符全部写入最终的输出文件中。在浮点数与字符串之间的转化过程中,若使用 sprintf 函数,速度会很慢,应用手动写的转化函数 ftoA 后,速度明显得到提高。

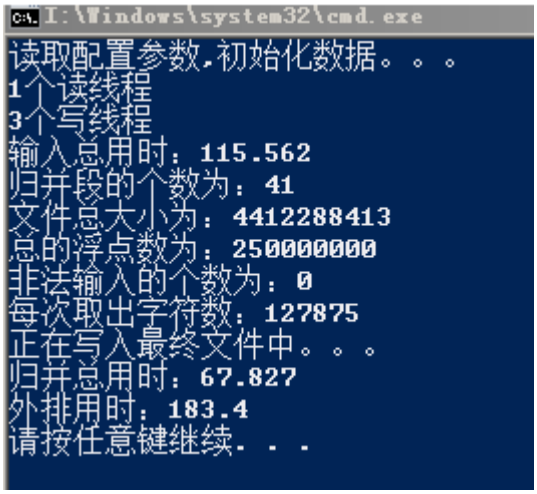
在 ftoA 转化函数中,可以使用二分查找来确定小数点的位置,可以将速度由 O(N)降低到 O(logN)。此外,在转化过程中,浮点数容易损失精度,因此需要在此浮点数第 11(要求保留 10 位)位有效数字加上 5,以保证精度。

四．结果

1. 实验测试环境：

机器系统：Windows 7 旗舰版
处理器：Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz
安装内存（RAM）：12.0GB
系统类型：64 位操作系统
编程工具：Microsoft Visual Studio 2012

2. 在 Microsoft Visual Studio 2012 的 release 模式下编译运行后，输出结果如图（3）所示：



图（3）

3. 不同数据规模的实验结果对比如图（4）所示：

实验数据	文件大小	最快运行时间	最慢运行时间	平均运行时间
100万浮点数	24.3MB	0.767s	0.889s	0.833s
5000万浮点数	922MB	26.902s	29.541s	27.805s
2.5亿浮点数	4.10GB	183.4s	316.546s	253.538s

图（4）

4. 程序运行过程中，内存使用峰值达到 530M。实验结果与镇霖的对比后，发现完全一致。

五．讨论

1. 以文本方式和以二进制方式读写文件的区别：

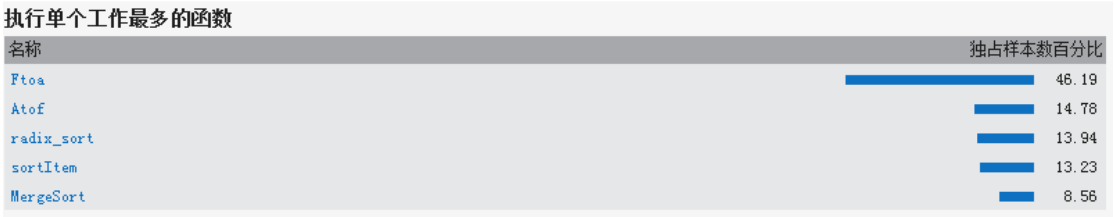
在读取文件和写入的过程中，发现分割的小文件的字符总和要大于原始文件，经查找资料发现，ifstream 类中的 open 函数在 Windows 下默认的打开方式问文本文件方式，而文本文件读写时，每遇到一个“\n”，它将其转化成“\r\n”，然后再写入文件；当文本读取时，它每遇到一个“\r\n”将其反变化为“\n”，然后送到读缓冲区。而二进制读写时，不存在任何转换，直接将写缓冲区中数据写入文件。故当打开文件时，需指明以二进制文件的形式打开。

2. 内部排序算法的对比和选择：

在实现外部排序的过程中，我主要使用了库函数 sort、快速排序、堆排序和基数排序四种内部排序算法对每次读到缓冲区中的数据进行排序。经对比发现：第一，经过四步优化（优化划分枢轴，优化不必要的交换，优化递归，优化小数组排序）后的快速排序的速度与库函数 sort 的速度相差无几；第二，堆排序可以增加每一个有序的临时文件的长度，假设堆的大小为 M，则可以产生大小为 2M 的顺串，减少访问外存的次数，节约时间，但其排序速度较慢；第三，针对浮点数排序而言，当数据量大于百万的时候，基数排序的速度明显优于快速排序等其他排序，故最终选择基数排序。

3. 手动实现 ftoA 函数的优化效果对比：

在优化的过程中，使用了 Microsoft Visual Studio 2012 的代码性能分析，未优化归并函数中的 sprintf 时，各函数运行时间比例如图（5）所示：



图（5）

手动实现 ftoA 函数后，各函数运行时间比例如图（6）所示：

图（6）