

## 2.5 亿个浮点数的外部排序报告

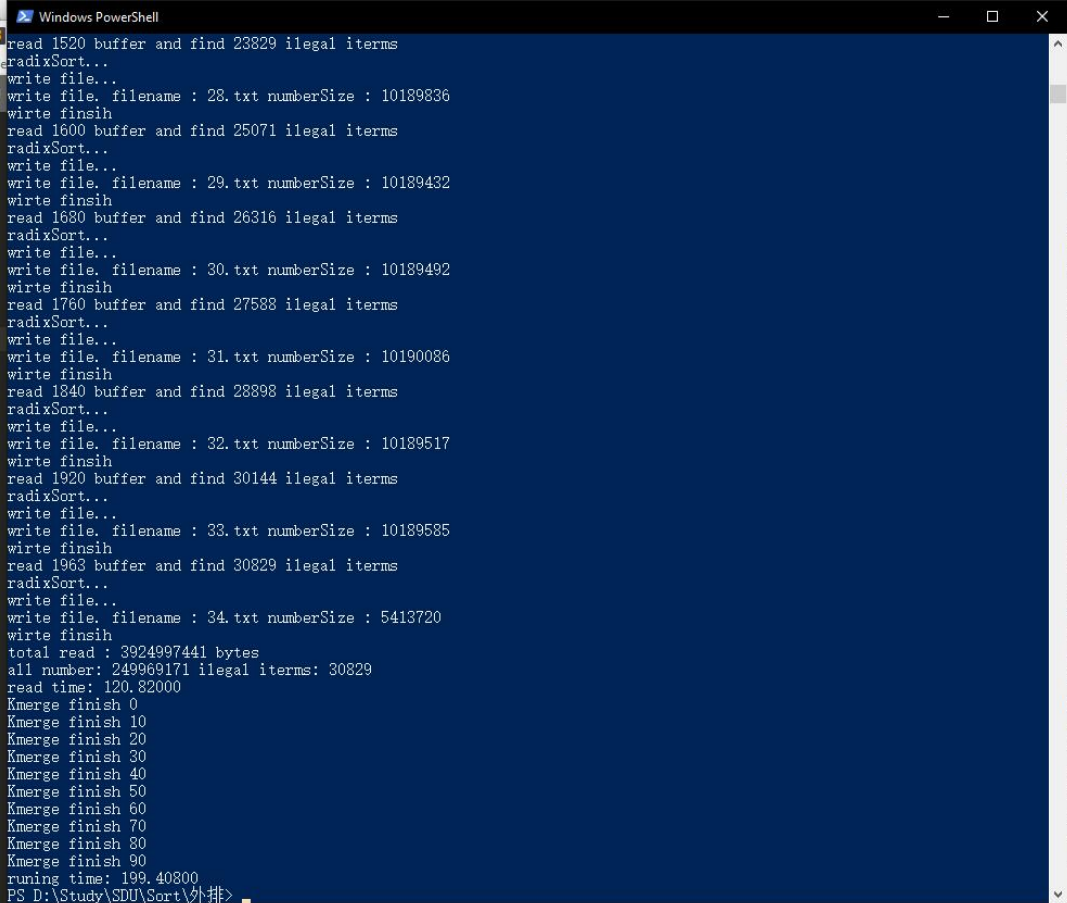
李绩成

2017 年 12 月 5 日

## 一、摘要

针对 2.5 亿个格式不确定的 IEEE754 标准下的浮点数排序问题，我的思路是通过单一线程使用 fread/fwrite 二进制分块读取大文件，然后通过自己实现 atof()，加速将读入的字符串转化为 double 的过程，并基于基数排序将得到的无序的 double 数组转化为多个有序的顺串，分别保存为单独的小文件，而后使用败者树多路归并多个小文件合并为一个大文件，最终完成整个排序。

经过测试在 Intel Core i7-4500U 1.8GHz 的个人 laptop 上的运行时间是 199.4s,如图所示：



```
Windows PowerShell
read 1520 buffer and find 23829 ilegal iterns
radixSort...
write file...
write file. filename : 28.txt numberSize : 10189836
write finsih
read 1600 buffer and find 25071 ilegal iterns
radixSort...
write file...
write file. filename : 29.txt numberSize : 10189432
write finsih
read 1680 buffer and find 26316 ilegal iterns
radixSort...
write file...
write file. filename : 30.txt numberSize : 10189492
write finsih
read 1760 buffer and find 27588 ilegal iterns
radixSort...
write file...
write file. filename : 31.txt numberSize : 10190086
write finsih
read 1840 buffer and find 28898 ilegal iterns
radixSort...
write file...
write file. filename : 32.txt numberSize : 10189517
write finsih
read 1920 buffer and find 30144 ilegal iterns
radixSort...
write file...
write file. filename : 33.txt numberSize : 10189585
write finsih
read 1963 buffer and find 30829 ilegal iterns
radixSort...
write file...
write file. filename : 34.txt numberSize : 5413720
write finsih
total read : 3924997441 bytes
all number: 249969171 ilegal iterns: 30829
read time: 120.82000
Kmerge finish 0
Kmerge finish 10
Kmerge finish 20
Kmerge finish 30
Kmerge finish 40
Kmerge finish 50
Kmerge finish 60
Kmerge finish 70
Kmerge finish 80
Kmerge finish 90
runing time: 199.40800
PS D:\Study\SDU\Sort\外排>
```

## 二、问题简介

编写一个可以对“海量”浮点数进行排序的程序 Sort.exe——读取文件路径和其他配置信息，将“输入文本文件”里面的浮点数从小到大排序，然后输出到另外一个“输出文本文件”中。完成后，用户双击 Sort.exe，程序即自动加载同目录下的 Sort.param，而后开始运行。Sort.param 里面放着“输入”和“输出”的文本文件（如.txt）的路径及其他信息。

### 三、程序设计与实现

#### 1) fread 将文件分块读入内存

思路：由于文件过大，无法一次性读入内存，因此使用 fread 将文件以二进制的形式读入内存之后，再以 8 位为单位，将读入的二进制解析为 char 类型，并通过分析读入的 char 类型找出非法条目，去除非法题目，保留合法条目并将合法条目转化为 double 类型，以待下一步处理。

#### 以二进制方式打开文件：

```
fp = fopen(rfile, "rb");
```

#### 读入一个 buf 的文件：

```
rl=fread(buf+overflow, sizeof(char), BUF_SIZE, fp);
```

此处，因为存在一个浮点数跨越两个块的情况，所以在处理每一个块的时候都会判断是否读到了 0x0a 或者 0x00（最后一个块），如果没有读到说明当前块中只含有该浮点数的部分数值，剩余数值需要从下一个块中获取。

因而这里设置了一个溢出量“overflow”用来表示上一个块中没有读完的字符，并将这些字符重新添加到 buf 的头部，同时下一个块在 buf 的存放位置向后推移 overflow：

```
if(*(cur-1)!='\n') {  
    overflow=len;//当前块中残缺的数值长度  
    for(i=0;i<overflow;++i) buf[i]=*(start+i);  
    if(rl!=BUF_SIZE) {  
        rdbytes+=overflow;  
        number=aTof(start,&isNum,len);  
        if(isNum) sortNum.push_back(number);  
        else nonumber++;  
    }  
}else {  
    overflow=0;  
}
```

#### 2) 将读入的字符数组转化为 double

思路：该过程所消耗的时间在整个过程中所占的比例是比较大的，所以为

了加速整个排序过程，需要自己实现一个 atof 函数，用以加速转换的过程。

首先，需要确定取到的这一个条目是否为非法条目。而后，针对合法的条目，先判断这个条目是否存在指数，如果存在指数，则先将指数计算出来，然后判断这个条目是否存在先导‘0’，并将先导‘0’去除（我对先导‘0’的定义是在第一个有效数字之前的‘0’），然后逐个分析字符，转化为相应的十进制数。

最后，为了后面处理的方便，也为了提高精度，在 atof 函数执行的过程中，还会进行四舍五入操作。而四舍五入的方法就是：在当前获得的有效数字的个数达到 10 的时候，判断下一位是否还是数字，如果是，则根据四舍五入的规则判断是否进位。

### 判断条目是否合法：

```
while(l<len){
    if(isDigit(npstr)||isAlpha(npstr)) ++npstr;
    else if(*npstr=='E' || *npstr=='e'){
        npstr++;
        se=(*npstr++);
        while(isDigit(npstr)) e=e*10.0+((*npstr++)-'0');
        l=(npstr-buf);
        e=(se=='-')?-e:e;
        break;
    }
    else break;
    ++l;
}
*legalNum=(l==len)?1:0;
```

当  $l \neq len$  的时候意味着，当前条目为非法条目。

### 累乘求整数部分：

```
while(isDigit(npstr))
    number=number*10.00+((*npstr++)-'0');
```

### 整数部分四舍五入：

```
if(isDigit(npstr)) number+=((( *npstr++)-'0')/5);
```

此处通过计算机除法的性质快速的完成四舍五入的操作。

**求小数部分:**

```
ev+=dpow[++pi]*(double)(( *npstr++)-'0');
```

此处通过预处理出 $10^{-k}$ 的值保存在 dpow 数组中，因此在求小数的过程中可以直接使用 dpow 加速处理速度。

**小数部分四舍五入:**

```
if(isDigit(npstr)) ev+=((( *npstr++)-'0')/5)*dpow[pi];
```

3) 获得 double 数组之后需要将其转换为顺串

**思路:** 由于 double 在计算机中的存储是 64 位的，和 long long 的位数相同，同时 double 的 64 位中高 12 位表示符号和指数，低 52 位表示格式化之后小数点后面的值。

因此，可以使用基数排序的方式，将 double 强制转化成 long long 并且将其按照 16 位一组分为 4 组，将每一个 double 视为一个 65536 ( $1 < 16$ ) 进制的数，并对每一个位进行基数排序。

由于负数最高位符号位是 1，正数符号位是 0，所以基数排序之后，正数都在负数之后。同时负数是从大到小排序的，正数是从小到大排序的。因此在 n 个数 (g 个负数) 的基数排序之后，正数的最终序号  $index=index+g$ ，负数的最终序号  $index=n-index-1$ 。

**将 double 强制转换成 long long:**

```
for (int i = 0; i < n; i++) a[i] = *(LL*)&nums[i];
```

**计算正确的序号 index:**

```
if (k == groups - 1)
    index=(a[i]<0)?(n-index-1):(index+negatives);
```

**long long 强制转化为 double:**

```
for (i = 0; i < n; ++i) nums[i] = *(double*)&t[i];
```

#### 4) 将顺串保存到小文件中

**思路：**这个过程和最后的写入大文件类似，都是一次磁盘 I/O，但是这一次 I/O 是不需要保存为可以阅读的文本格式的，也就是说，可以直接将 double 的二进制编码写入文件中，后面需要读入的时候在将读取到的二进制编码以 64 位的格式进行解析，转化成 double。

在获取顺串的时候，由于不知道读入的每一块中含有多少个有效条目，因此该处使用了 vector 数组来保存获取到的 double，这就意味着在将顺串写入小文件的时候，需要将 vector 写入文件。

另外，由于每一次写入都会触发一次磁盘 I/O，因此为了提高处理速度，每一个小文件中应该包含多个块，每一次写入都将多个块的数据写入文件中，这样也可以保证小文件的数目不会过多。

(程序中小文件的命名是从 10.txt 开始，逐渐加一。)

#### 将 vector 以二进制方式写入文件：

```
fwrite(&sortNum[0],1,len*sizeof(double),fp);
```

#### 5) 待初始文件被转换成多个小文件之后，将每个小文件的一部分读入内存

**思路：**由于内存限制，因此不能一次性将所有小文件全部读入内存中。此时我决定对每一个小文件采取分块读取的思想。由于一共只有 512MB 内存，因此每个小文件可以读取 15MB 左右的块，大约是 1,900,000 个浮点数，因此每一次从一个小文件中取出 1,900,000×8 Bytes 大小的块。

#### 以二进制的方式打开所有小文件：

```
for(int i=0;i<fcot;){
    sf[i]=fopen(filename,"rb");
    if(sf[i]==NULL) printf("ERROR\n");
    arr[i]=new double[arrSize+5];
    ++i;
    if(i%10==0) {
        filename[0]++;
        filename[1]='0';
    }
    else filename[1]++;
}
```

```
}
```

fcot 是小文件的个数，filename 是一个 char 数组其中保存着小文件的文件名。

### 从每一个小文件中获取一个块：

```
double **arr=new double*[fcot];  
int arrayElementsCount[fcot];  
for(int i=0;i<fcot;++i)  
    arrayElementsCount[i]=fread(arr[i],  
    sizeof(double),arrSize,sf[i]);
```

arr 保存的是从每个小文件中取出的块，arrayElementsCount 保存的是这个块的大小。

### 6) K-Merge 败者树的运用以及 double 转字符数组

思路：在读取到每个小文件的部分块之后，需要使用败者树将这些数据合并为一个文件。在一个文件的当前块被处理完毕之后，需要重新从这个文件中取出新的一块，直到这个文件整个处理完毕。

首先，从小文件中获得的是 double 的二进制表示，再次解析成 double 之后，放入败者树中，并维护败者树。为了跟更加快速的实现多路并归，我基于数组实现了自己的败者树，而败者树的叶子节点的个数取决于上一步所产生的小文件的数目。

然后从败者树中取出的最小值需要先转化为字符数组，然后存放在一个 buffer 中，等这个 buffer 装满之后再将这个 buffer 整个写入最终的大文件中。同时，为了更块的实现 double 向 char 类型的转换，并且这种转换是被指定精度和有效数字个数的，需要自己实现一个 dtoa 函数。

而我实现这个函数的思路就是，二分枚举这个 double 的十进制指数，直到找到这个使得当前 double 的值乘以当前枚举的指数的十进制幂之后，其绝对值属于[1.0,10.0)。同时，为了处理的方便，对于负数，可以将其二进制的符号位置为 0 (与 0x7FFFFFFFFFFFFFFF)，这样负数后面的处理其实和正数是一样的。

### 败者树的建立：

```
void createLoserTree(double* arr[],int &fcot, int* &root,
```

```
double* &leaf)///败者树
{
    for (int i = 0; i < fcot; ++i) leaf[i] = arr[i][0];
    leaf[fcot] = minDouble;//minkey
    for (int i = 0; i < fcot; ++i) root[i] = fcot;
    //有 k 个叶子节点
    //从最后一个叶子节点开始，沿着从叶子节点到根节点的路径调整
    for (int i = fcot - 1; i >= 0; --i)
        Adjust(fcot, root, leaf, i);
}
```

此处，arr 存放着 fcot 个小文件当前读入的块，而 fcot 表示小文件的个数，root 表示败者树中的非叶子节点，leaf 则表示败者树中的叶子节点。

### 败者树的维护：

```
void Adjust(int &fcot, int* &root, double* &leaf, int i)
{
    //控制 ls[] 的下标
    int t = (i + fcot) / 2;//第一个非叶子结点的下标、第二个。。。
    //控制 b[] 的下标
    int s = i;
    for (; t > 0; t /= 2){
        if (leaf[root[t]]<leaf[s]) swap(s, root[t]);
    }
    root[0] = s;
}
```

其中 i 表示第 i 个叶子节点。

### dtoa 函数中枚举 double 的十进制指数：

```
a. IF double<1.00
    int l=0,r=308;
    while(l<=r){///二分找十进制的指数
        mid=(l+r)>>1;
```



```

        tmp=number*lpow[mid];
        if(tmp>=10.00){
            r=mid-1;
        }else if(tmp<1.00){
            l=mid+1;
        }else break;
    }
b. IF double>=10.00
    int l=0,r=308;
    while(l<=r){ ///二分找十进制的指数
        mid=(l+r)>>1;
        tmp=number/lpow[mid];
        if(tmp>=10.00){
            l=mid+1;
        }else if(tmp<1.00){
            r=mid-1;
        }else break;
    }

```

其中 `lpow[k]` 保存的是  $10^k$ ,  $k \in [0, 308]$ 。同时通过测试发现,如果要乘以  $10^{-k}$ ,即将 `double` 的十进制小数点向左移动  $k$  位,除以  $10^k$  会比乘以  $10^{-k}$  更加高效。另外由于存在精度的损失,所以我在转换之前为每一个数添加了一个精度损失量,这个精度损失量是小于当前浮点数中十进制精度下的最低有效位的单位值的,比如在这个问题中,我设置的精度损失量 (`0x00000000000000400`) 就要比  $10^{-10}$  还要小,这样并不会影响最终的结果,也可以保证 `6.200000000E-308` 不会输出为 `6.199999999E-308`。

**判断当前块是否处理完,如果是,则重新取新的一块,直到文件末尾:**

```

if (findindex[s] < arrayElementsCount[s]){
    arr[s][0] = arr[s][findindex[s]];
}else{
    findindex[s]=0;
    if(arrayElementsCount[s]==arrSize)
        arrayElementsCount[s]=fread(arr[s],

```

```

        sizeof(double), arrSize, sf[s]);
    else arr[s][0]=maxDouble;
}

```

其中，`findex[s]`表示着第  $s$  个小文件当前已经处理完的 `double` 的数量，而 `arrayElementCount[s]`表示第  $s$  个小文件当前所读入的块包含的 `double` 的数量。因为每一个文件最后一块的大小不可能刚好的 `arrasize`（此处设置为 1,900,000）的的整数倍（因为前面对小文件大小的设置导致每一个小文件所包含的 `double` 的个数不可能是 1,900,000 的整数倍），所以当发现当前块的大小不等于 `arraSize` 的时候就可以判断文件已经读完了。此时将多路并归的第  $s$  路后面的值都设置为最大（0x7FF0000000000000）。

**将 buffer 写入最后的大文件中：**

```

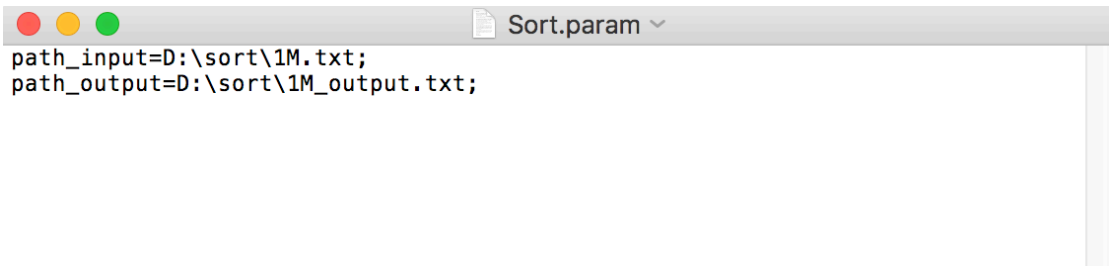
if(writeBytes+curlen<bufSize){ //buf 还有空间容纳
    wstart=dtoa(wstart,leaf[s]);
    writeBytes+=curlen;
    writebuf[writeBytes++]=0x0a;
}else{//buf 已经装满
    //写入文件
    fwrite(&writebuf[0],sizeof(char),writeBytes,ffile);
    //更新 buf 的已占用空间，将其置为 0
    writeBytes=0;
    wstart=&writebuf[0];
    wstart=dtoa(wstart,leaf[s]);
    writeBytes+=curlen;
    writebuf[writeBytes++]=0x0a;
}

```

## 四、结果

最终，程序在 i7-4500U 处理器上的性能为 2.5 亿个浮点数最终完成时间为 199.4s，其中读文件、`atof`、基数排序、写小文件一共耗时 120.82s，而读小文件、`Kmerge`、`dtoa`、写大文件一共耗时约 79s。

附：程序中所使用的 `Sort.param` 文件的配置：



```
path_input=D:\sort\1M.txt;
path_output=D:\sort\1M_output.txt;
```

## 五、讨论

针对这个问题，以及我目前解决问题方法的设计与思路，我觉得可以通过以下两个方法在一定条件下加快排序的速度。

### 1. 多线程

读大文件+atof、基数排序都是两个十分消耗时间的操作，因此可以设计一个多线程的架构，在读大文件的同时也执行基数排序，这样可以在一定程度上加快排序的速度。

### 2. 数据的随机性

在数据足够随机并且数据量足够大的情况下，可以发现写入大文件的数据中存在一个规律，就是相同次幂的数据是集中在一起的，并且不同次幂的数据在指数上的表现是：文件最前部分的次幂是+308，而后逐渐减少，直到+0，然后-1、-2...-308、-307、-306...-2、-1、+0、+1、+2...+308。这样利用数据的规律性，在数据量足够大并且随机的情况下，可以通过直接获得double的十进制指数。比如，如果当前程序预测的十进制指数  $K=-308$ ，而实际的double的十进制指数为-307，那么可以将这个double乘以 $10^{308}$ 之后结果会大于10，那么就可以将K减少一，这样就获得了当前double的实际十进制值是-307，在之后很长的一段时间内-307都将是正确的十进制指数，直到出现一个-306。