Dr. Fulton’s Dissertation on Quicksort for Huge-data

Author : Douglas Shaw Fulton

# 前言

第一部分(与论文主体无关):

首先，讨论某种形式的快速排序的代码，找出最基本的程序

在基本程序的基础上进行优化

优化的过程实际上是一系列的数据测试，通过把数据规模转换为时间从而比较算法的优化结果

数据记录

## 快速排序的讨论

我们知道快速排序最重要的部分是分区操作，分区函数一般如下：

partiction(int l,int r)//设要讨论的数组为a

set pivot；

while true：

first（i，j）//找到一对i，j来定位要交换的值

do（i，j）//

partiction函数可以按照递归方式定义,partition(l,r)=first(i,j)+do(i,j)+partition(i+1,j-1),这是尾递归的,可以重写成循环的形式.

其中比较重要的部分是，我们怎样处理在first(I,j)之后的代码,其次,怎样的first(I,j)代码会影响到do(I,j)代码。首先，我们不讨论具体的first，但是我们知道，在first之后i，j处于两种状况：①i>j ②i<j 没有第三种情况,并且我们知道如果i<j那么就应当交换i,j位置的元素然后进入下一次循环判断,因此退出的情况就是i>j.

如果i>j,如果①i>r,也就是i=r+1超出正常下标,说明所有的元素都<pivot,那么j=r;有i=j+1,**此时的处理应当是寻找一个比pivot小的元素重新分区(错误:应当坚持一个pvoit的原则)** ,但由于在选取元素为数组的某一值的情况下,不会所有元素都<pvoit,所以这种情况可以忽略②j<l,对应的,所有元素>=pvoit,i=l,也有i=j+1;这时,如果分区是失败的(即彻底未分过区)则可以重新选择基准元素,否则认为分区成功③i,j的下标正常,那么a[i]>=pvoit,a[i-1]<pvoit,a[j]<pvoit;由于i>j,且a[i-1]是第一个满足a[j]关系的,所以一定有i=j+1;此时应当从j处分区,下两次排序发生在(l,j)和(j+1,r).

分区函数具有尾递归的性质,可以转换为循环,但是条件是使用新的变量来记录参数.

所以,java形式的快速排序代码如下:

qsort(l,r)

{

if(r-l+1<=1)return;

else

{

pos=partition(l,r);

qsort(l,pos);

qsort(pos+1,r);

}

}

## 快速排序的优化

以下是几种设想,①基准元素既然可能发生全部大于,全部小于的情况,不如取三个元素中间的一个,这样就避免了I,j范围超标的问题;②如果数组重复性很大,则每一次分区时,将<Pivot ,>=Pivot换成 <Pvoit,=Pvoit,>Pvoit,三部分,其中中间部分只需要在>=Pvoit的部分再次分区即可 ③在一定的元素个数时候,使用插入排序或者冒泡排序,这种方法能大幅改进性能,④由于qsort明显像一个尾递归函数,因此使用栈+循环的模式改进

## 优化的程序设计

1. **元素取中**

三个元素时寻找中间一个,同时交换它们的,位置;

若end-start>=r,start+r/2=<(start+end)/2<=end-r/2,为使三个坐标互不相同,r=4,仅当end-start>=4时进行三元素取中

a,b,c

i j k

if(\*j>\*i)

{

swap(\*i,\*j);//I,j有序

}

if(k>j)//nothing

else if(k>i)swap(k,j);

else {TEMP=arr[k];arr[k]=arr[j];arr[j]=arr[i];arr[i]=TEMP;}

pvoit=arr[j];

插入的源代码:

*TEMP*=start;//<end,若end-start>=r,start+r/2=<(start+end)/2<=end-r/2,为使三个坐标互不相同,r=4

if(end-start>=4)

{

if(ARR[leftIndex]>ARR[pvPos])

{

TEMP=leftIndex;

}

if(ARR[rightIndex]>=ARR[leftIndex]);//do nothing

else if(ARR[rightIndex]>=ARR[pvPos])

{

TEMP=rightIndex;

}

else{

TEMP=pvPos;

}

}//三元素法

pvPos=*TEMP;*

1. **再次分区**

在常规的partition之后,调用right-partition(l,r,pvoit)

即

int j=l;while(arr[j]==pvoit)++j;

for(int i=r;i>=l;--i)

{

if(a[i]==pvoit){swap(arr[i],arr[j]);j++}

}

插入的源代码:

//通过再次分区来减小rightIndex

j=rightIndex;**while**(j<end&&*ARR*[j]==pvoit)++j;

**for**(i=end-1;i>j;--i)

**if**(*ARR*[i]==pvoit)

{

*TEMP*=*ARR*[i];*ARR*[i]=*ARR*[j];*ARR*[j]=*TEMP*;

**while**(j<end&&*ARR*[j]==pvoit)

++j;

**if**(j==end)**break**;

}

rightIndex=j;//j==end-1,end通过

1. **在一定元素的时候使用其他排序(easy)**

使用CONTROL参数,

if(r-l+1<=CONTROL){other-sort(r,l+1)};

1. **栈+循环改进递归**

**qsort(l,r)**

**stack<int> s1,s2;**

**s1.push(l);**

**s2.push( r);**

**while(!s1.empty())**

**{**

**l=s1.pop();**

**r=s2.pop();**

**if…**

**else**

**{**

**midIndex=partition..**

**s1.push(l);s2.push(midIndex);**

**s1.push(midIndex+1);s2.push(r);**

**}**

**}**

### 准备测试程序

int[] ARR=new int[40000000];

int SIZE=0;

1. 数据使用一个随机生成的数组

**public** **static** **void** getRandomData(**int** size)

{

Random r=**new** Random();

**for**(**int** i=0;i<size;++i)

*ARR*[i]=r.nextInt();

*SIZE*=size;

}

1. 计时器使用

long start=System.*currentTimeMillis*();

qsort();

long end=System.*currentTimeMillis*();

Util.p(start-end);

1. 测试数组是否已经排序了

public static boolean isSorted(int[] arr,int start,int end)

{

if(end-start<=1)return true;

int i=start;

while(i<end-1&&arr[i]==arr[i+1])++i;

if(i<end-2)//至少有三个不相同元素才能确定是否排序

{

if(arr[i]<arr[i+1])//<=

while(++i<end-1)

{

if(arr[i]>arr[i+1])return false;

}

else//>=

while(++i<end-1)

{

if(arr[i]<arr[i+1])return false;

}

}

return true;

}

## 快速排序的优化记录

实际上以上所有的工作都只是整个任务的一小部分,我认为接下来—实践这些优化才是最重要、最挑战耐力和智力的时候.首先,设计的测试方案是”未优化---优化一个项模型—全部优化—标准库”.

### 数据

B1:未优化 B2.1:优化① B2.2:优化② B2.3:优化③ B2.4:优化④ B3:全部优化 B4:标准库

A:100 1000 10000 50000 100000 1000000 5000000 10000000 20000000 30000000 40000000(上限)

B1: 0 2 3 11 21 307 1413 2971 6114 9562 13928

B2.1: 0 1 2 11 22 262 1416 2995 6269 10298 12750(数据足够大,优化有所体现)

B2.2:0 2 2 13 33 338 1532 3136 6598 10168 13577

B2.3: 1 1 2 8 47 248 1245 2553 5431 8289 11329 (CONTROL=20)

B2.4: 2 5 28 78 131 1257 6106 5330 11103 16612 22981

B4:5 0 8 10 13 14 171 932 1911 4014 6146 8730

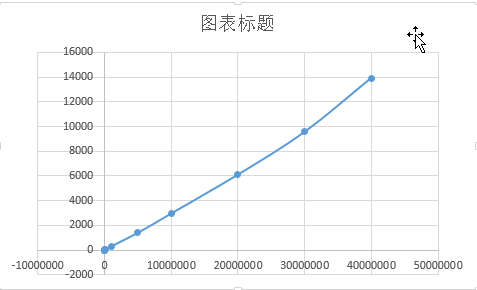
结论真是令人大吃一惊,即所谓的优化,只有③起了显著作用,①也仅在数据较大时有所体现,②几乎未优化,而④反而降低了性能

最优化程序采用③,①

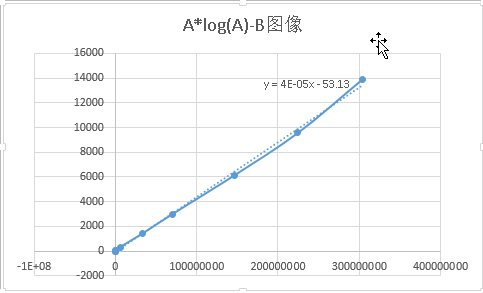
B3结果: 0 1 1 8 20 299 1335 2793 5930 8918 12060

未优化数据组

A-B1图像:



A\*log(A)-B1图像:逼近公式B1=4\*10-5A\*logA-53.13



未优化的源代码:

public static void quickSort(int start,int end)

{

if(end-start<=1)return;// <=1个元素则不排序

else

{

int startPos=start,endPos=end-1;//为了使递归转化为循环而进行的操作

int leftIndex=0,rightIndex=0;

int pvPos=start;

int pvoit/\*基准元素\*/=ARR[pvPos];

int i=startPos,j=endPos;

while(true)

{

// Util.p(Util.arrayString(ARR, 0, SIZE));

if(endPos<startPos)

{

leftIndex=startPos;

rightIndex=endPos+1;

break;

}

else if(endPos==startPos)

{

if(ARR[startPos]<pvoit)

{

leftIndex=startPos+1;

}

else

{

leftIndex=startPos;

}

rightIndex=leftIndex;

break;

}

while(i<=endPos&&ARR[i]<pvoit)++i;

while(j>=startPos&&ARR[j]>=pvoit)--j;

if(j<startPos)//所有元素>=pvoit,分区有可能失败了

{

if(startPos==start&&endPos==end-1)//分区失败

{

j=startPos;

while(j<=endPos&&ARR[j]==pvoit)++j;//寻找一个大于pvoit的新元素

if(j>endPos)//所有元素=pvoit,数组有序了

{

leftIndex=startPos;

rightIndex=endPos+1;

break;

}

else

{

pvoit=ARR[j];//重设pvoit,重新开始

startPos=j;

i=startPos;j=endPos;

continue;

}

}

else//分区进行过,且本次分区startPos~endPos都是满足>=pvoit的

{

leftIndex=rightIndex=startPos;

break;

}

}

else if(i==j+1)

{

leftIndex=rightIndex=i;

break;

}

else //j>i

{

//swapArr(list, i, j);

TEMP=ARR[i];

ARR[i]=ARR[j];

ARR[j]=TEMP;

startPos=++i;

endPos=--j;

}

}

// Util.p(Util.arrayString(ARR, 0, SIZE));

//Util.pn(list);Util.p();

if(leftIndex<end)

quickSort(start, leftIndex);

if(rightIndex>start)

quickSort(rightIndex, end);

}

}

### 总结陈词:我相信这是一次有益于科学的探究,并且也是如此.这个案例将作为典范载入我学生时代科学生涯的史册,并且将永远被效仿.

## 评论区

有一个定律说,代码越简单越容易理解.确实如此,但也不尽然.我认为复杂的不是代码,而且代码也不应当被视为复杂或简单的,代码是为了实现算法而写的,换句话说,it will be what it should be.而且,结构复杂、分支众多的代码比简单的代码性能更快,因此,简单代码的唯一优势就是看着舒服(也不一定如此,但和复杂代码相比确实如此).因此,不要纠结代码的复杂与否,it will be what it should be,反而,我们应当关注伪代码(那些用来描述算法的代码)的结构.代码要多复杂就多复杂吧,我不认为这很重要.