分布式与并行计算实验

---

project:

枚举、归并、快速排序的串行与并行算法比较

院系：人工智能学院

姓名：石睿

学号：211300024

班级：分布式与并行计算-2023 秋季学期

邮箱：211300024@smail.nju.edu.cn

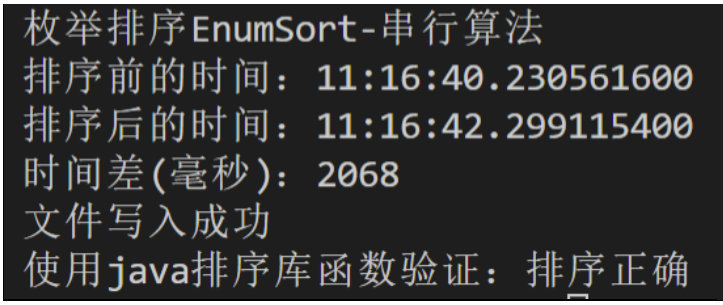
实验时间：2023.12.12

一、 实验进度

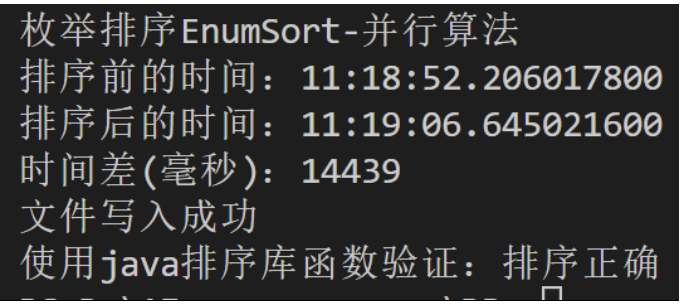
我已经完成了本次课程 project 的所有内容，也即分别实现快速排序、枚举排序、归并排序三种排序方法的串行与并行算法。

二、 实验结果

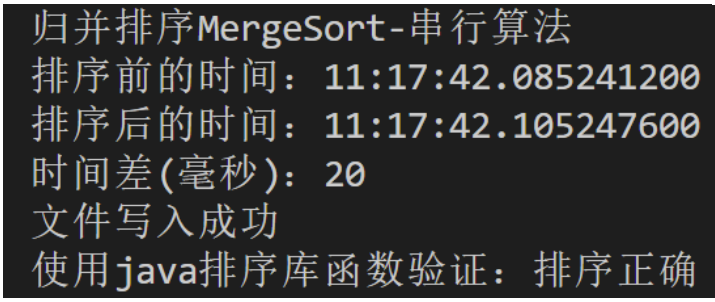
本次实验使用 java 完成三种串行和并行算法，**算法未优化运行结果**如下图所示（此时并行算法不额外引入任何串行部分）。各算法均运行多次，取出最符合均值的运行结果。同时由于使用了 git 版本控制，每次运行后 orderi.txt 并没有出现 modified（M）标志也就是多线程运行结果符合预期（无 interleave，是可串行化调度）。同时也借助 java 排序库函数 Arrays.sort()来验证串行、并行算法正确性。具体实践细节将在第三部分详细阐释。



1.1： 实现串行的枚举排序算法

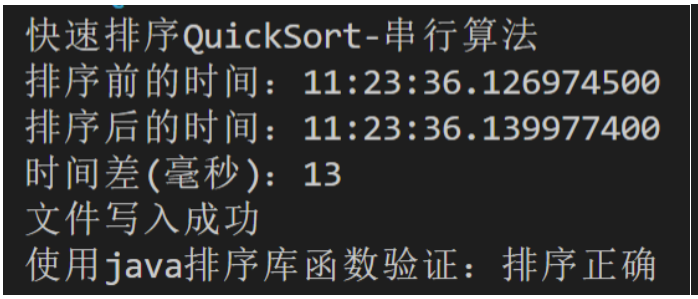


1.2： 实现并行的枚举排序算法

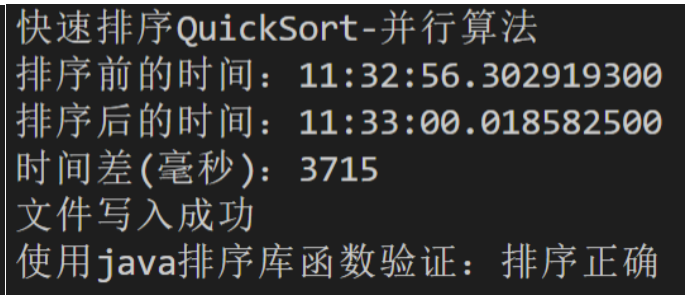


2.1： 实现串行的归并排序算法

2.2： 实现并行的归并排序算法



3.1： 实现串行的快速排序算法



3.2： 实现并行的快速排序算法

三、 技术要点

① 枚举排序 EnumSort

1.1 串行算法

枚举排序算法核心思想是对每个元素统计小于它的所有元素个数，从而得到该元素在序列中的位置。如 a[1]和 a[2],…,a[n]比较，记录比 a[1]小的个数 k，最后存放在 b[k]的位置上（下标从 0 开始）一共 n（n-1）次，复杂度下限是 O(n^2)。其算法伪代码如下所示。

```
Begin
fori=1 to n do
  (1)k=1
  (2)forj=1 to n do
      if a[i] > a[j] then
        k=k+1
      end if
     end for
  (3)b[k]= a[i]
end for
End
```

运行多次后对运行时间取平均值，可以得到串行的枚举排序所需时间约为 2000ms（2.0s）（此时复杂度是 O(n^2)）。

1.2 并行算法

上述枚举排序不是一个好的串行算法，但是其并行算法很好设计，从伪代码的角度来看加个 para-do 就行！因为本枚举排序串行算法是松耦合的。可以轻易地解耦成多个 cpu，且各个 cpu 之间可以各自做事，无需频繁交互！类似的，如果串行算法中 for 很多，则被转化成良好的并行算法的可能性会大很多。并行枚举排序的伪代码如下所示。

PRAM-CRCM：有SM，不需要广播

```
Begin
(1)P0播送a[1]…a[n]给所有Pi
有SM的不用广播，但如COW就需要，通讯复杂度O（n^2）
n个处理器，每个处理器要接收n个数据

(2)for all Pi where 1≤i≤n para-do
Review：严格来说，这个for应该不写para-do，直接用for all P_i就可以啦。
但考虑到二者本质等价性，也还算可以喽
    (2.1)k=1
    【每一个处理器对应的a[i]分别运行以下代码，O(n)】
    (2.2)for j= 1 to n do
            if (a[i] > a[j]) or(a[i] =a[j] andi>j)
            then
              k = k+1
            end if
          end for
(3)P0收集k并按序定位
COW需要计算好的k，通讯O（n），只回收k个数字即可，n*1

End
```

以下为通过 java 具体实现代码。此时增加了同步障 barrier，初始化定义时传入一个整形变量 parties，表示只有到 parties 个线程同时运行到 barrier.await()处后，各线程才能共同运行下去；增加了类 ES，其有一个构造函数 ES(int[] arr,int[] result, int target, CyclicBarrier barrier)和运行函数 public void run()。

在经过多次试验中发现，不加同步障 barrier 和加入同步障 barrier 结果相同。可能是因为 main 线程返回 result 之前，所有线程完成对 result 数组的赋值更新。但在逻辑上为了保持可串行化调度，是必须要加上 barrier。否则在 main 线程返回 result 前，线程没有完成对 result 赋值，对应位置值全部为 0（java 对未初始化数组的定义）。

```
public static int[] enum_sort(int[] arr,int whe){
    int result[] = new int[(int)arr.length];
    CyclicBarrier barrier = new CyclicBarrier(arr.length+1);//【必须k_m个线程都到达才能做！】
    for(int target : arr){//【可以并行化设计，一个进程对应一个元素小于它个数的计算】
        if(whe == 0){//【串行】
            int sum = 0;
            for(int item : arr){
                if(item < target)
                    sum++;
            }
            result[sum] = target;
        }
        else if(whe == 1){//【并行】
            ES new_thread;
            new_thread = new ES(arr,result,target,barrier);
            new_thread.start();
        }
    }
    try {
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e) {
        //TODO Auto-generated catch block
        e.printStackTrace();
    }
    return result;
}
```

运行多次后对运行时间取平均值，可以得到符合设计逻辑（加入 barrier）的并行的枚举排序所需时间约为 14000ms（14s）。

对比串行和并行的枚举排序算法，可以得到并行算法约为串行算法时间消耗的 7 倍（在不进行任何优化的时候）。造成性能很差的原因是 barrier 和线程数量太多（新开 arr.length）。考虑对并行枚举排序算法优化，其中 barrier 不可以删除。可以使部分 arr 中元素调用并行 enum_sort，进而减小新开线程数量以及在同步障 barrier 的等候时间！

② 归并排序 MergeSort

2.1 串行算法

归并排序算法核心思想原数组进行折半划分，分别对两个子序列递归排序 Merge_Sort 后再进行两子序列的合并 Merge。复杂度下限是 O(n·logn)。其算法伪代码如下所示。

```
MERGE_SORT(A, p, r)
  if p < r
    q = (p + r) / 2
    MERGE_SORT(A, p, q)
    MERGE_SORT(A, q + 1, r)
    MERGE(A, p, q, r)

MERGE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  Let L[0.. n1] and R[0..n2] be new arrays
  for i = 0 to n1 - 1
    L[i] = A[p + i]
  for j = 0 to n2 - 1
    R[j] = A[q + j]

  i = j = 0;
  for k = p to r
    if L[i] <= R[j]
      A[k] = L[i]
      i = i + 1
    else
      A[k] = R[j]
      j = j + 1
```

运行多次后对运行时间取平均值，可以得到串行的归并排序所需时间约为 18ms（此时复杂度是 O(n·logn)）。

2.2 并行算法

我首先尝试使用方根划分技术（Valiant 算法）来完成 Merge 的并行化，它的复杂度为 O(loglogn)，但由于划分后边界条件过多就没有继续完成了。后续实验采用对数划分技术来对 Merge 操作进行并行化，复杂度为 O(logn)。对数划分技术的伪代码如下。

```
Begin
(1)j(0)=0; j(k(m))=n;
A中主元在A中的位置（后续更新成b_i·logm大于等于多少元素个数）
(2)for i=1 to k(m)-1 par-do
    求rank(b_i·logm:A);
    j(i) = rank(b_i·logm:A);
此时A是排好序的（merge的前提），下标即可代A的主元
endfor
(3) for i=0 to k(m)-1 par-do
    B_i = (b_i·logm +1 ,..., b_(i+1)·logm);
    A_i = (a_j(i) +1 ,..., a_(j(i+1)));
各个线程调用串行mergesort算法对划分出来的A、B小段分别merge
endfor
End
```

同枚举排序的并行算法设计，方根划分也需要引入同步障。必须在所有新创建的线程(k_m)个和 main 线程同时到达 return ans(merge 后的结果元祖)前，才可以由 main 线程 return ans。否则，main 线程在新创建线程还未完成自己对 ans 中规定位置的 merge，则会返回很多的 0!

```
CyclicBarrier barrier1 = new CyclicBarrier(k_m+1);//【必须k_m个线程都到达才能做！】
```

```
if(process == k_m-1){ //main线程的同步障设计，在main不需要创建新线程后，挂起
    try {//等待其他线程完成对ans的赋值！
        barrier1.await();
    } catch (InterruptedException | BrokenBarrierException e) {
        //TODO Auto-generated catch block
        e.printStackTrace();
    }
```

运行多次后对运行时间取平均值，可以得到并行的归并排序所需时间约为 4000ms (4s)。

对比串行和并行的枚举排序算法，可以得到并行算法约为串行算法时间消耗的 60 倍（在不进行任何优化的时候）。造成性能很差的原因是 barrier 和线程数量太多。考虑对并行枚举排序算法优化，其中 barrier 不可以删除。可以在 k_m (A、B 中划分段数)小于一定常数

值的时候调用串行算法 MergeSort.merge，进而减小新开线程数量以及在同步障 barrier 的等候时间！如在划分段数小于 1000 时候直接并行 merge，运行时长如下所示。

```
归并排序MergeSort-并行算法
排序前的时间：22:23:24.765447600
排序后的时间：22:23:25.015504600
时间差(毫秒)：250
文件写入成功
```

③ 快速排序 QuickSort

3.1 串行算法

快速排序的核心步骤有以下三步：

3.1.1 从原始数组中随机选择主元（因为原数据分布未必均匀，取第一个、最后一个、中间元素作为主元都未必在期望上保持平均划分原数组【中位数的中位数也可以滴！保证平衡】）

3.1.2 依主元划分成左右两个数组，左侧小，右侧大

3.1.3 分别递归调用两个子数组

对应伪代码如下所示。

```
总体算法
Begin
  call procedure quicksort(data,1,n)
End

函数1:
procedure quicksort(data,i,j)
Begin
(1)if (i<j) then
  (1.1)r = partition(data,i,j)
  (1.2)quicksort(data,i,r-1);
  (1.3)quicksort(data,r+1,j);
end if
End

函数2：划分函数
procedure partition(data,k,l)
Begin
(1)pivo=data[l]# 此时主元的选择是数组最后一个元素
(2)i=k-1
(3)for j=k to l-1 do
    if data[j]≤pivo then
      i=i+1
      exchange data[i] and data[j]
    end if
   end for
(4)exchange data[i+1] and data[l]
(5)return i+1
End
```

书上提供的伪代码 partition 思路不是很明确，具体实践时采用原算法课讲的 partition 想

法--通过双指针法来实现（复杂度也是 O(n)）。

运行多次后对运行时间取平均值，可以得到串行的快速排序所需时间约为 13ms（此时复杂度是 O(n·logn)）。

3.2 并行算法

有两种方法可以实现对 QuickSort 的并行化：以下分别介绍其细节并实现。

3.2.1 对每次调用的 quicksort( ) 使用两处理器递归排序

其核心想法：(1.2)quicksort(data,i,r-1);使用一个处理器； (1.3)quicksort(data,r+1,j);——使用另外一个处理器。只做了 quicksort 的直接串行化并发，但真正做事的 partition 并没有并发，而只实现了模式的并发。

对应的时间复杂度为 O(n)（可使用主定理推导出来）。此时必须限制创建线程的个数，否则会到达 java 允许创建新线程数量上限。运行多次后对运行时间取平均值，可以得到串行的快速排序所需时间约为 3700ms(3.7s)。

```
else{
    //【报错：线程开的过多会报错】
    //【逻辑错误：上一次没排好，main着急去创建下一次的线程就不行！】
    if(after_p[0]-1 - Left < 3000 || Right - after_p[1]+1 < 3000){
        quickSort(arr, Left, after_p[0]-1, whe);
        quickSort(arr, after_p[1]+1, Right, whe);
    }
```

```
procedure para_quicksort(data,i,j,m,id)
i和j还是初始、截止位置
m：剩余可用处理器的取log后次数（初始可用个数2^m）
id：指定处理器的id
Begin
(1)if (j-i)≤k or m=0 then
  (1.1)P_id call quicksort(data,i,j)
else
  (1.2)P_id: r=partition(data,i,j)
  (1.3)P_id send data[r+1,j] to P_id + 2^(m-1)-1
信息传走：把划分的后一半数组给新的处理器
  (1.4)para_quicksort(data,i,r-1,m-1,id)
qicksort：原来负责划分的处理器，仍然处理前一半数组
  (1.5)para_quicksort(data,r+1,j,m-1,id+2^(m-1)-1
quicksort：新处理器，后一半
----1.4和1.5在本次并发优化中，可以并行来做！
  (1.6)P_id+2^m-1 send data[r+1,j] back to P_id
信息传回：从新处理器把处理后数组拿回来
end if
End
```

3.2.2 对 partition 的并行化

对 partition 进行并行化的核心想法有以下四步，首先进行符号规定：

待排序的序列为(A1,…,An)，处理器 Pi 保存元素 Ai。fi 存当前主元所在的处理器号、根：主元 root、左子树：小于主元的元素 LC[1:n]、右子树：大于主元的元素 RC[1:n]

step1: 所有处理器把他们的处理器号写入 root。

step2: A_root 是主元，并把 root（主元所在处理器号）复制给每一个处理器 i 的 fi

step3: 每个处理器对 Ai 和 Afi 做比较，把 Ai 写入 LC_fi 或 RC_fi【所有处理器的 fi 值相同】

step4: 由于 Ai 写入 LC_fi 或 RC_fi 也是在 CRCW 上，最后一个写入的就是新的主元

```
Begin
(1) For each processor i do
(1.1) root=I
(1.2) fi=root
(1.3) LCi=RCi=n+1
End

(2) Repeat for each processor i!= root do
If (Ai<Afi) or (Ai=Afi and i<fi) then
(2.1) LCfi=i
(2.2) if i=LCfi then exit
      else fi=LCfi
      endif
else
(2.3) RCfi=i
(2.4)  if i=RCfi then exit
       else fi=RCfi
       endif
Endif
Endrepeat
End
```

此时算法期望复杂度为 O(logn)。

④ 各算法未优化&优化后运行时间（保持一定并发程度）

未优化

	枚举排序	归并排序	快速排序
串行	1660ms	12ms	11ms
并行	16000ms	4000ms	3700ms

优化后（保证一定“并行”程度（单核处理器只能用并发模拟并行））

	枚举排序	归并排序	快速排序
串行	1660ms	12ms	11ms
并行	16000ms	445ms	344ms

此时分别对归并排序和快速排序引入规模相等的串行部分（递归子数组长度小于 10 采用串行计算），仍保留极高的“并行”（并发）程度，但可以显著降低运行时耗。枚举排序由于没有递归，若引入串行部分计算意义不大。

考虑到并行算法多线程在单核处理器中的并行开销，并行算法性能会比串行算法有所下降。三种排序对应串行和并行的比较在前文已分析，此处就不再赘述。