

10. R中的并行计算II

罗翔宇

中国人民大学统计与大数据研究院

此课件内容基于Norman Matloff编著的《数据科学中的并行计算》（CRC Press, 汪磊、寇强译）

上周复习

- ▶ 用于R中并行的snow包
- ▶ `library(parallel)`
- ▶ `makeCluster(4)` 用于产生snow集群
- ▶ `clusterExport(cls, "x")` 将数据或者函数发送给worker
- ▶ `clusterApply(cls, ichunks, doichunk)` 采用轮询的方式指导worker对分配的块任务进行计算
- ▶ `clusterApplyLB()` 将块任务进行动态调度, 尽可能实现负载均衡
- ▶ `stopCluster(cl2)` 计算任务完成后, 停止对应snow集群

例子：所有可能回归，改进版本

- ▶ 如果我们熟悉线性回归的话，我们知道估计量 $\hat{\beta} = (X'X)^{-1}X'Y$ ，其中 X 的第一列均为1，表示回归中的截距项
- ▶ 由于我们对不同的预测变量集合均用了 $\text{lm}()$ 函数，其中就需要重复利用矩阵相乘，这些计算量其实可以通过提前计算好 $(X'X)$ 和 $X'Y$ 再取子矩阵避免
- ▶ 比如 \tilde{X} 由 X 的第(1,3,4)列构成，那么 $\tilde{X}'\tilde{X}$ 就是 $X'X$ 的第(1,3,4)列和第(1,3,4)行构成，因此不用重新计算，只需提取子矩阵即可

例子：所有可能回归，改进版本

► 改进代码

```
#####  
###improved version  
  
#here the first column of x corresponds to the intercept term  
#find the adjusted R square  
linregadjr2 <- function(x, y, xpx, xpy){  
  bhat <- solve(xpx, xpy)  
  resid <- y - x %*% bhat  
  r2 <- 1 - sum(resid^2) / sum((y-mean(y))^2)  
  n <- nrow(x)  
  p <- ncol(x) - 1  
  1 - (1-r2)*(n-1)/(n-p-1) #adjusted R2  
}  
  
#improved version of dolpset  
dolpset_v2 <- function(onepset, x, y, xpx, xpy){  
  ps <- c(1, onepset + 1)  
  x1 <- x[,ps]  
  xpx1 <- xpx[ps, ps]  
  xpy1 <- xpy[ps]  
  ar2 <- linregadjr2(x1, y, xpx1, xpy1)  
  n0s <- ncol(x) - length(ps)  
  c(ar2, onepset, rep(0, n0s))  
}
```

例子：所有可能回归，改进版本

► 改进代码

```
dochunk_v2 <- function(psetsstart, x, y, xpx, xpy, allcombs, chunksize){  
  ncombs <- length(allcombs)  
  lasttask <- min(psetsstart+chunksize-1, ncombs)  
  t(sapply(allcombs[psetsstart:lasttask], do1pset_v2, x, y, xpx, xpy))  
}
```

例子：所有可能回归，改进版本

► 改进代码

```
snowapr_v2 <- function(cls, x, y, k, reverse=F, dyn=F, chunksize=1){  
  # cls: snow cluster  
  # x: predictor matrix, one column corresponds to a predictor  
  # y: response vector  
  # k: maximum size of predictor variable sets  
  # reverse: TRUE if we reverse the iteration order  
  # dyn: TRUE if we use dynamic assignment  
  # chunksize: size of a chunk  
  
  require(parallel)  
  p <- ncol(x)  
  x <- cbind(1, x)  
  xpx <- crossprod(x, x) #equivalent to and slightly faster than t(x) %**% x  
  xpy <- crossprod(x, y)  
  
  allcombs <- genallcombs(p, k)  
  ncombs <- length(allcombs)  
  
  clusterExport(cls, "do1pset_v2")  
  clusterExport(cls, "linregadjr2")  
  
  #set the starting indices  
  tasks <- if(!reverse) seq(1, ncombs, chunksize)  
    else seq(ncombs, 1, -chunksize)  
  
  if(!dyn){  
    out <- clusterApply(cls, tasks, dochunk_v2, x, y, xpx, xpy,  
                        allcombs, chunksize)  
  }else{  
    out <- clusterApplyLB(cls, tasks, dochunk_v2, x, y, xpx, xpy,  
                         allcombs, chunksize)  
  }  
  Reduce(rbind, out)  
}
```

例子：所有可能回归，改进版本

► 改进代码

```
snowtest_v2 <- function(cls, n, p, k, chunksize=1, dyn=F, rvrs=F){  
  gendata(n, p)  
  snowapr_v2(cls, x, y, k, rvrs, dyn, chunksize)  
}
```

► 运行结果

```
> c2 <- makeCluster(2)  
> system.time(snowtest(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE))  
用户 系统 流逝  
5.81 29.53 35.74
```

```
> system.time(snowtest_v2(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE))  
用户 系统 流逝  
5.78 29.39 35.31
```


例子：所有可能回归，改进版本

► 运行结果，尝试不同的chunksize

```
> system.time(snowtest(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE, chunksize = 20))
用户 系统 流逝
0.25 1.44 8.58
> system.time(snowtest_v2(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE, chunksize = 20))
用户 系统 流逝
0.25 1.66 1.95
> system.time(snowtest(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE, chunksize = 50))
用户 系统 流逝
0.27 0.62 7.85
> system.time(snowtest_v2(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE, chunksize = 50))
用户 系统 流逝
0.14 0.67 0.91
```

例子：所有可能回归，改进版本

► 运行结果，尝试不同的chunksize

```
> system.time(snowtest(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE, chunksize = 20))
用户 系统 流逝
0.25 1.44 8.58
> system.time(snowtest_v2(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE, chunksize = 20))
用户 系统 流逝
0.25 1.66 1.95
> system.time(snowtest(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE, chunksize = 50))
用户 系统 流逝
0.27 0.62 7.85
> system.time(snowtest_v2(c2, 10000, 20, 3, dyn = TRUE, rvrs = FALSE, chunksize = 50))
用户 系统 流逝
0.14 0.67 0.91
```

► 关于分块大小：如果分块太小，就需要处理更多的块，因而会产生更多的开销；如果块太大，运行快要结束时会遇到负载均衡问题

- 可以通过在一两个问题上做实验，决定一个较好的块大小
- 使用随时间变化的块大小

multicore包

- ▶ parallel包是由R的两个包，snow和multicore构成的
- ▶ multicore包必须在一个多核机器上运行，且被限制在Unix家族的操作系统，比如Linux和Mac OS X，在这样的平台上multicore包比snow包性能更好
- ▶ 性能提升的源头：multicore包中的并行函数mcapply()会利用Unix族的系统中叫做fork()的系统调用
- ▶ fork()可以将一个进程分叉为多个子进程，而且这些子进程含有原进程中所有变量一模一样的拷贝。因此在所有可能回归的例子中，分出的子进程自带xpx,xpy，而不用传输所需的时间，其性能会高于snow()
- ▶ mcapply()中的参数mc.cores来选择内核个数

例子：并行计算距离

- ▶ 假设有两个数据集，分别有 m 和 n 个观测，我们需要计算 mn 个组合之间的距离
- ▶ 在很多机器学习的实例中都需要进行这样的计算。比如，我们想用身高、年龄来预测体重
- ▶ 有一个训练集含有信息(身高、年龄、体重)，一个预测集(身高、年龄)
- ▶ 对于预测集中的一个人，我可以在训练集中找到身高、年龄和他“相近”的个体，通过计算训练集中这些个体的平均年龄，来作为预测值
- ▶ 如何定义“相近”，就需要计算两个集合间样本的距离了

例子：并行计算距离

► 代码

```
#install.packages("pdist")
library(pdist)
library(parallel)

#divide 1:m into chunks with approximate size "chunksize"
npart <- function(m, chunksize){
  splitIndices(m, ceiling(m/chunksize))
}

#calculate samples in "ichunk"
dochunk <- function(ichunk, x, y){
  as.matrix(pdist(x[ichunk, ], y))
}
```

- 需要安装pdist包，用于利用C语言高效计算两个集合样本间的距离。as.matrix将其转化为矩阵
- splitIndices将数组1:m尽量均匀地分为若干块，每块的大小约为chunksize

例子：并行计算距离

► 代码（展现了pdist和splitIndices的功能）

```
> a <- matrix(rnorm(20), 5, 4)
> b <- matrix(rnorm(16), 4, 4)
> pdist(a, b)@dist
 [1] 3.373154 2.776329 3.552311 1.248351 1.871680 3.524712 1.235919 2.812059 2.039737 3.988303 1.720160
[12] 2.779948 2.751270 2.614185 3.917880 3.615190 3.325737 2.384787 3.421149 2.181060
attr(,"csingle")
[1] TRUE
> splitIndices(10, 3)
[[1]]
[1] 1 2 3

[[2]]
[1] 4 5 6 7

[[3]]
[1] 8 9 10
```

例子：并行计算距离

► 代码

```
#calculate samples in "ichunk"
dochunk <- function(ichunk, x, y){
  as.matrix(pdist(x[ichunk, ], y@dist)
}

snowpdist <- function(cls, x, y, dyn = F, chunksize = 1){
  nx <- nrow(x)
  ny <- nrow(y)

  ichunks <- npart(nx, chunksize)
  clusterExport(cls, "pdist")
  if(!dyn){
    dists <- clusterApply(cls, ichunks, dochunk, x, y)
  }else{
    dists <- clusterApplyLB(cls, ichunks, dochunk, x, y)
  }

  tmp <- Reduce(rbind, dists)

  tmp
}

genxy <- function(m, n, p){
  x <- matrix(runif(m*p), ncol = p)
  y <- matrix(runif(n*p), ncol = p)
}
```

例子：并行计算距离

► 结果

```
> system.time(pdist(x,y))
```

用户	系统	流逝
12.77	1.00	13.78

```
>
```

```
> c2 <- makeCluster(2)
```

```
> system.time(snowpdist(c2, x, y, dyn = TRUE, chunksize = 500))
```

用户	系统	流逝
121.50	22.57	151.78

```
> stopCluster(c2)
```

- 可能的原因，pdist通过底层C代码实现，已经非常的高效；若利用并行计算，有通信上的开销（传输一个较大的矩阵），反而时间变慢了。若能增加核的个数（比如8个），并增加数据的维度，则速度会有较大的提升

foreach包

- ▶ R中另一个流行的并行化工具是**foreach包**，其通过CRAN下载，主要用于提高for循环的速度
- ▶ 并行化方式简单，比如将for更换为foreach，并添加一个运算符 %dopar%
- ▶ 用户还需要指明要运行的平台，即foreach包中的用于后端，它可以是snow包、multicore包、**doParallel包**等

```
> #install.packages("doParallel")  
> library(doParallel)  
> registerDoParallel()  
> getDoParWorkers()  
[1] 3  
> registerDOSEQ()  
> getDoParWorkers()  
[1] 1
```

- ▶ getDoParWorkers()在默认情况下在Windows上创建3个workers，在Linux, Mac系统上创建一半核数的workers

foreach包

- ▶ 使用foreach要小心，并不能够认为用了foreach一定具有速度上的提升
- ▶ R中的for其实经过了优化，大多情况下，它的速度可能远远快于foreach

```
> system.time(for(i in 1:n){s = s+i})
用户 系统 流逝
  0    0    0
> n <- 10^5
> s <- 0
> system.time(for(i in 1:n){s = s+i})
用户 系统 流逝
  0    0    0
> s <- 0
> registerDoParallel(cores = 2)
> system.time(foreach(i = 1:n) %dopar% {s = s+i})
用户 系统 流逝
26.26  7.47 34.89
```

- ▶ 想说明的是，我们应该对foreach()格外注意，**避免认为只要使用了这个包，就可以轻易地平行；或者只要让代码并行化就可以获得最大的速度提升。**

共享内存范式：基于R的简单介绍

- ▶ 两种并行范式：共享内存范式(共享内存)、消息传递范式(分布式内存、通过网络进行通信)
- ▶ **共享内存**指所有的处理器都共享同一块内存地址空间
- ▶ 共享内存范式：多个并行的进程通过存取机器中它们所共同使用的内存(RAM)单元来互相通信
- ▶ 例子：多核机器、图像处理单元(Graphics processing unit, GPU)等
- ▶ Rdsmp包可以从R语言的级别来完成共享内存并行化
- ▶ 针对某些应用，共享内存编程比其他的R并行包运行地快得多

共享内存代码的简洁

- ▶ 消息传递范式：变量X在进程1中，变量y在进程2中，将变量X复制给变量y（利用Rmpi包）
 - ▶ `mpi.send.Robj(x, tag=0, dest=2) #进程1`
 - ▶ `y <- mpi.recv.Robj(tag=0, source=1) #进程2`
- ▶ 共享内存范式：变量X和y都被共享，程序员仅需要编写
 - ▶ `y <- x`
- ▶ 使用共享内存可以极大简化代码、减少混乱，因此与消息传递环境相比，可以更快地编写和调试我们的程序

Rdsm包

- ▶ Rdsm包的好处：运行在共享内存中，且使用R来编程。R不能在R编程级别提供多线程，然而Rdsm包给R带来了多线程。缺点为只能在Linux或者Mac上进行安装。
- ▶ 矩阵乘法的例子（需要安装bigmemory包）

```
mmultithread <- function(u, v, w){  
  require(parallel)  
  myidxs <- splitIndices(nrow(u), myinfo$nwkrks)[[myinfo$id]]  
  w[myidxs, ] <- u[myidxs, ] %*% v[ , ]  
  0 #do not return w for its heavy cost on time  
}
```

- ▶ myinfo\$nwkrks表示线程个数，myinfo\$id表示当前线程的id
- ▶ 不返回所计算的值。返回计算结果会消耗较多的时间，而且由于w是共享变量，也没必要返回

Rdsm包

► 矩阵相乘代码

```
test <- function(cls){
  #initialize Rdsm
  mgrinit(cls)

  #set shared variables
  mgrmakevar(cls, "a", 6, 2)
  mgrmakevar(cls, "b", 2, 6)
  mgrmakevar(cls, "c", 6, 6)

  #assign values
  a[ , ] <- 1:12
  b[ , ] <- rep(1, 12)

  #export function to threads
  clusterExport(cls, "mmulthread")

  #run
  clusterEvalQ(cls, mmulthread(a, b, c))
  print(c[ , ])
}

library(parallel)
c2 <- makeCluster(2)
test(c2)
```

所得结果

```
> test(c2)
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]      8      8      8      8      8      8
[2,]     10     10     10     10     10     10
[3,]     12     12     12     12     12     12
[4,]     14     14     14     14     14     14
[5,]     16     16     16     16     16     16
[6,]     18     18     18     18     18     18
```

mgrinit() 用来初始化Rdsm包系统

mgrmakevar() 用来在共享内存中创建三个矩阵

使用snow包的clusterEvalQ()来开始线程

Rdsm包与普通运算的计时对比

► 普通代码

```
> library(Rdsm)
> library(bigmemory)
> n <- 2000
> m <- matrix(runif(n^2), n, n)
> system.time(m%*%m)
   user  system elapsed 
5.270   0.004   5.275
```

Rdsm包与普通运算的计时对比

► 使用Rdsm代码（调用四个线程）

```
> cls <- makeCluster(4)
> mgrinit(cls)
[[1]]
[1] 0

[[2]]
[1] 0

[[3]]
[1] 0

[[4]]
[1] 0

> mgrmakevar(cls, "msh", n, n)
> mgrmakevar(cls, "msh2", n, n)
```

```
> mgrmakevar(cls, "msh", n, n)
> mgrmakevar(cls, "msh2", n, n)
> msh[ , ] <- m
> clusterExport(cls, "mmulthread")
> system.time(clusterEvalQ(cls, mmulthread(msh, msh, msh2)))
   user  system elapsed 
0.001   0.000   1.516
```


共享内存能够带来性能优势

- ▶ 共享内存与消息传递的比较
- ▶ 从消息传递上，就能看出它做了很多数据复制的工作，有时候是大量的数据，这在很多情况下都是不必要的
- ▶ 下面我们将Rdsm包和利用消息传递范式的snow包进行速度对比

```
> snowmmul <- function(cls, u, v){  
+   require(parallel)  
+   idxs <- splitIndices(nrow(u), length(cls))  
+   mmulchunk <- function(idxchunk) u[idxchunk, ] %*% v  
+   res <- clusterApply(cls, idxs, mmulchunk)  
+   Reduce(rbind, res)  
+ }
```

共享内存能够带来性能优势

► 比较函数

```
> testcmp <- function(cls, n){  
+   require(Rdsm)  
+   require(parallel)  
+   mgrinit(cls)  
+   mgrmakevar(cls, "a", n, n)  
+   mgrmakevar(cls, "c", n, n)  
+   amat <- matrix(runif(n^2), n, n)  
+   a[ , ] <- amat  
+   clusterExport(cls, "mmulthread")  
+   print(system.time(clusterEvalQ(cls, mmulthread(a, a, c))))  
+   print(system.time(cmat <- snowmmul(cls, amat, amat)))  
+ }
```

共享内存能够带来性能优势

► 不同核数、不同矩阵大小下的速度对比

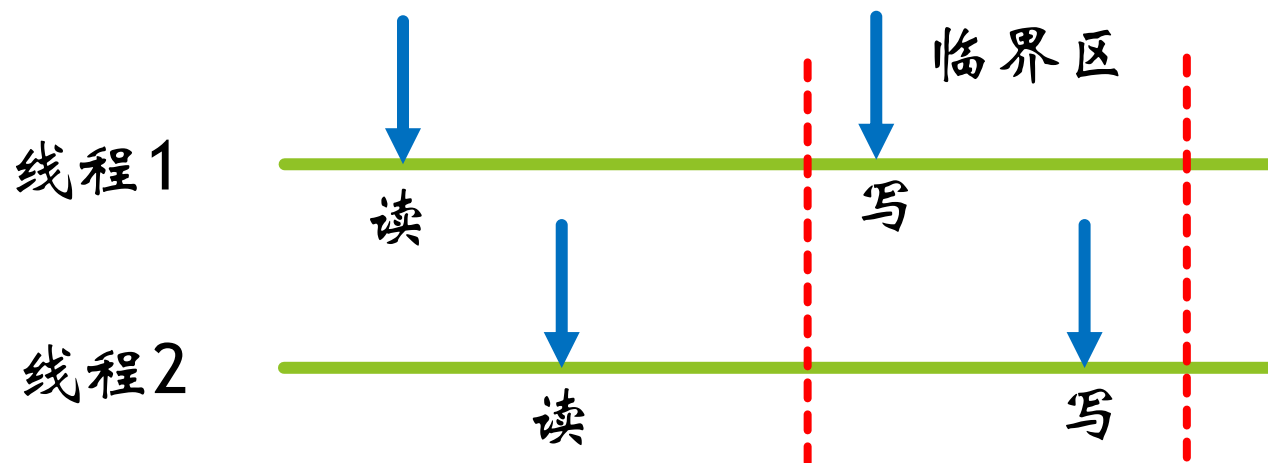
```
> cls <- makeCluster(4)
> testcmp(cls, 2000)
  user  system elapsed
0.001   0.000   1.522
  user  system elapsed
0.705   0.080   2.532
> stopCluster(cls)
> cls <- makeCluster(8)
> testcmp(cls, 3000)
  user  system elapsed
0.002   0.000   3.535
  user  system elapsed
3.026   1.385   8.325
> stopCluster(cls)
```

锁和屏障

- ▶ 竞争条件和临界区
- ▶ 一个场景：两个顾客都想预定指定日期的指定航班，他们几乎同时登陆了预订系统。每个顾客运行一个线程。假设这个航班只剩下一个座位。每个线程都会发现这个航班只剩下一个座位，因此每个线程都进入**临界区**（顾客的数据录入到指定日期的指定航班的代码区）。从而每个线程都会为顾客预定这个航班。
- ▶ 但是，一个线程会比另一个稍微先执行，第二个线程信息会覆盖前一个线程所写的信息。也就是，第一个顾客认为她成功预定好了，但其实是没有的。
- ▶ 这个临界区充满了被称为**竞争条件**的情形

锁和屏障

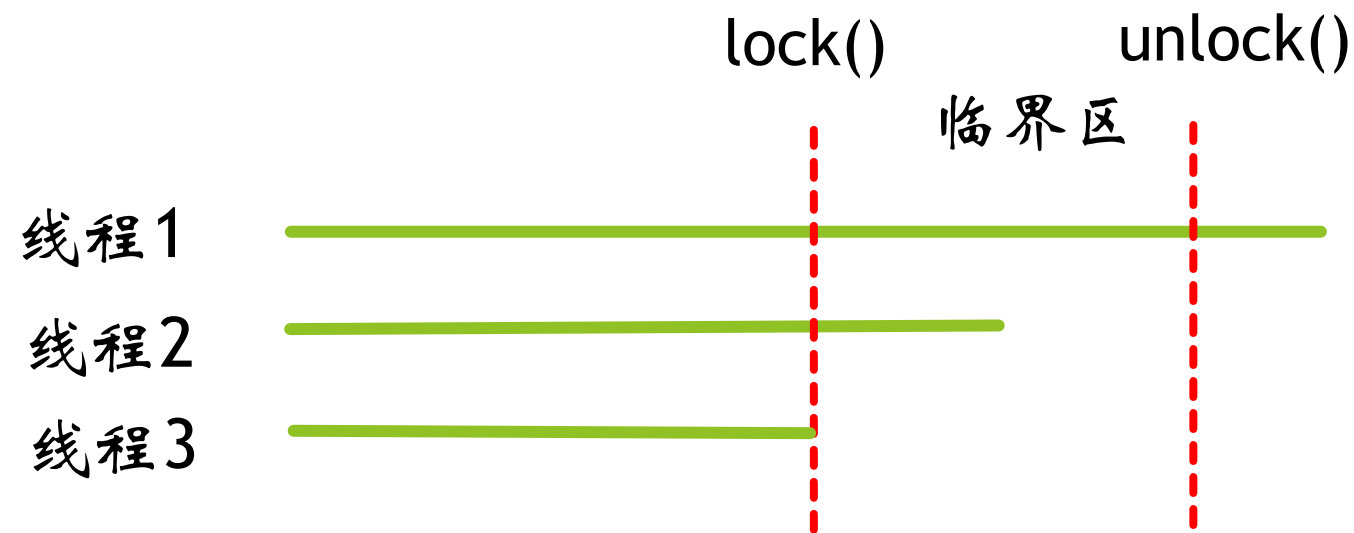
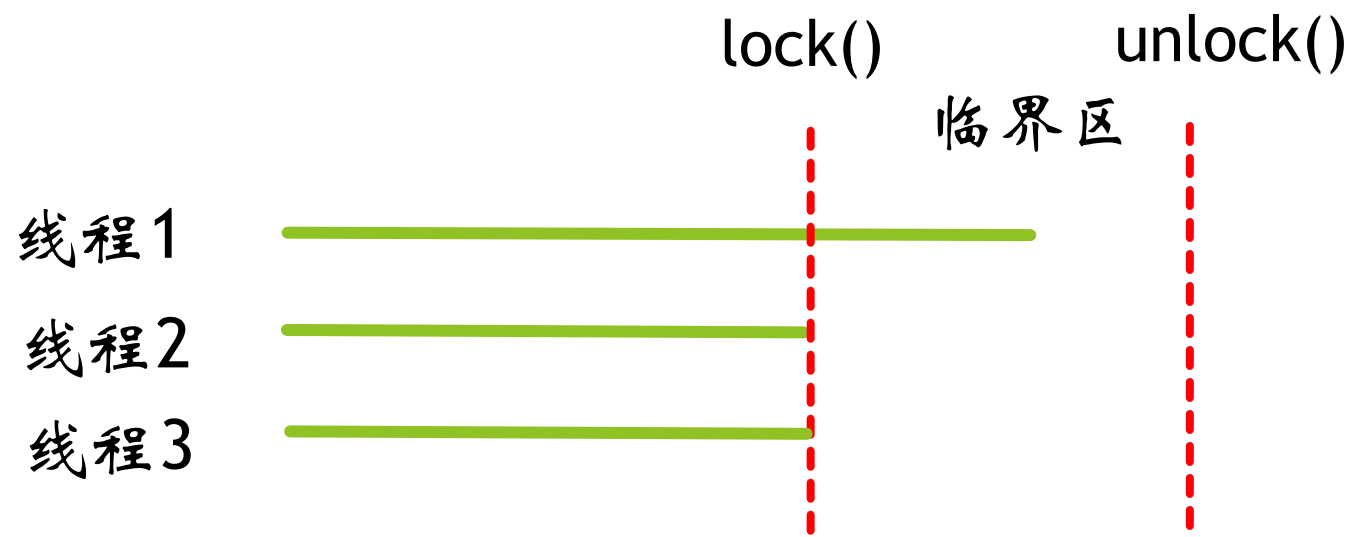
- ▶ 造成这种情况的原因是：每个线程的“读”（是否有位置）和“写”（录入顾客数据）是分开的。因此需要将读和写看成一个不可分割的组合，即原子化(atom)



- ▶ 为了避免竞争条件，需要一个机制，同一个时刻只有一个线程可以存取临界区，这被称为互斥。
- ▶ 通过利用锁变量以及调用函数lock()和unlock()可以实现

锁和屏障

- ▶ 在临界区的前后分别调用函数lock()、unlock()



锁和屏障

- ▶ 一个例子，如下函数不可靠，2个线程同时都试着让总数加1，它们会互相干扰。最终结果不为2000，而是比2000小很多。

```
> s <- function(n) {  
+   for(i in 1:n){  
+     tot[1,1] <- tot[1,1] + 1  
+   }  
+ }  
> library(parallel)  
> c2 <- makeCluster(2)
```

```
> clusterExport(c2, "s")  
> library(Rdsm)  
> library(bigmemory)  
> mgrinit(c2)  
[[1]]  
[1] 0  
  
[[2]]  
[1] 0  
  
> mgrmakevar(c2, "tot", 1, 1)  
> tot[1,1] <- 0  
> clusterEvalQ(c2, s(1000))  
[[1]]  
NULL  
  
[[2]]  
NULL  
  
> tot[1,1]  
[1] 1033
```

锁和屏障

► 原因可能为

- thread 1 reads tot[1,1], finds it to be 227
- thread 2 reads tot[1,1], finds it to be 227
- thread 1 writes 228 to tot[1,1]
- thread 2 writes 228 to tot[1,1]

锁和屏障

- 若我们加上lock()和unlock()函数

```
> s1 <- function(n){  
+   for(i in 1:n){  
+     rdsmlock("totlock")  
+     tot[1,1] <- tot[1,1] + 1  
+     rdsmunlock("totlock")  
+   }  
+ }  
> mgrmakelock(c2, "totlock")  
> tot[1,1] <- 0  
> clusterExport(c2, "s1")  
> clusterEvalQ(c2, s1(1000))  
[[1]]  
NULL  
  
[[2]]  
NULL  
  
> tot[1,1]  
[1] 2000
```

锁和屏障

- ▶ **屏障**用来同步所有**线程**的，和MPI_Barrier有相似的作用，只不过后者是用来同步进程的
- ▶ 在Rdsm包中，这个函数为barr()，当一个线程调用它的时候，这个线程会被阻塞，直到所有的线程都调用了这个函数