9. R中的并行计算I

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）

安装RStudio

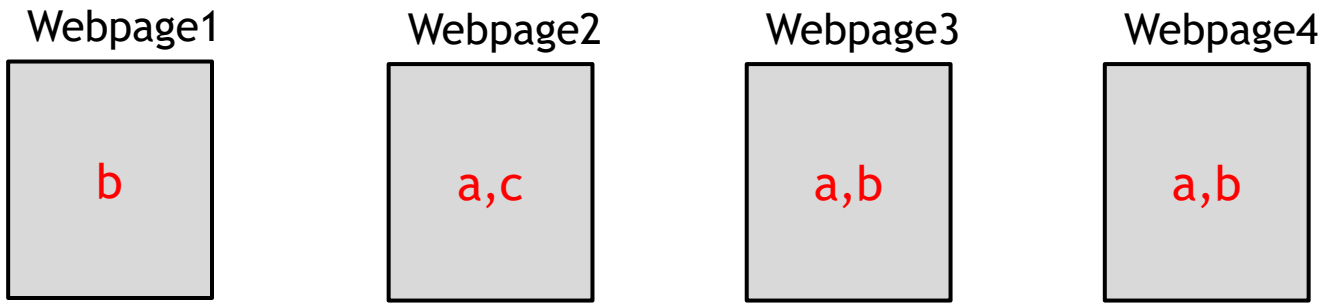
- https://rstudio.com/products/rstudio/

RStudio Desktop

	Open Source Edition	RStudio Desktop Pro
Overview	• Access RStudio locally • Syntax highlighting, code completion, and smart indentation • Execute R code directly from the source editor • Quickly jump to function definitions • Easily manage multiple working directories using projects • Integrated R help and documentation • Interactive debugger to diagnose and fix errors quickly • Extensive package development tools	All of the features of open source; plus: • A commercial license for organizations not able to use AGPL software • Access to priority support • RStudio Professional Drivers
Support	Community forums only	• Priority Email Support • 8 hour response during business hours (ET)
License	AGPL v3	RStudio License Agreement
Pricing	Free	$995/year

例子：相互网页外链

- 在分析网页流量数据的时候，一个重要问题是计算所有网页之间的、相互外链的平均数目



- 4个网页，两个网页都链接到同一网站的情况: (1-2,0), (1-3,1), (1-4,1), (2-3,1),(2-4,1),(3-4,2)
- 相互外链的平均数目为: (0+1+1+1+1+2) / 6 = 1

例子：相互网页外链

- 串行代码

```
mutoutser <- function(links){
  nr <- nrow(links)
  nc <- ncol(links)
  tot <- 0
  for(i in 1:(nr-1)){
    for(j in (i+1):nr){
      for(k in 1:nc){
        tot <- tot + links[i, k] * links[j, k]
      }
    }
  }
  tot / (nr * (nr-1) / 2)
}
```

- links是一个nr行nc列的矩阵
- links[i, k]表示网页i是否具有链接k，是的话为1，不是的话为0

例子：相互网页外链

- 探究此串行代码性能，我们用如下模拟

```
sim <- function(nr, nc){
  lnk <- matrix(sample(0:1, (nr*nc), replace = TRUE), nrow = nr)
  system.time(mutoutser(lnk))
}

sim(1000, 1000)
```

- 所得到的结果如下

```
   用户    系统    流逝
 48.73    0.02   49.16
```

  - 用户时间给出了当前R进程花费的CPU时间
  - 系统时间给出了操作系统代表当前进程花费的CPU时间，比如打开文件、输入输出、启动其它进程等
  - 流逝时间指从R进程开始所实际流逝的时间，用了49.16秒

例子：相互网页外链

- R中的显示for循环很慢。一个避免for循环的方法是向量化，这样可以享受到向量运算底层C代码带来的速度

旧版本

```
for(i in 1:(nr-1)){
  for(j in (i+1):nr){
    for(k in 1:nc){
      tot <- tot + links[i, k] * links[j, k]
    }
  }
}
```

改进1

```
for(i in 1:(nr-1)){
  for(j in (i+1):nr){
      tmp <- links[j, ] %*% links[i, ]
      tot <- tot + tmp
    }
  }
}
```

改进2

```
for(i in 1:(nr-1)){
    tmp <- links[(i+1):nr, ] %*% links[i, ]
    tot <- tot + sum(tmp)
}
```

例子：相互网页外链

- 向量化版本的函数

```
mutoutser1 <- function(links){
  nr <- nrow(links)
  nc <- ncol(links)
  tot <- 0
  for(i in 1:(nr-1)){
    tmp <- links[(i+1):nr, ] %*% links[i, ]
    tot <- tot + sum(tmp)
  }
  tot / (nr * (nr-1) / 2)
}

sim1 <- function(nr, nc){
  lnk <- matrix(sample(0:1, (nr*nc), replace = TRUE), nrow = nr)
  system.time(mutoutser1(lnk))
}

sim1(1000, 1000)
```

- 所得结果，仅用了4.28秒

```
用户 系统 流逝
2.84 1.44 4.28
```

- 若改为

```
> sim1(2000, 2000)
   用户   系统   流逝
23.45   9.74 33.21
```

- 则需要并行计算进行进一步加速

例子：相互网页外链

- 并行工具的选择：R语言中并行最流行的工具有snow包、multicore包、foreach包和Rmpi包。前两个工具是R中基础包parallel包的一部分
- 由于multicore包只能运行在Linux或Mac的平台上，不能在Windows上运行，因此我们主要讨论snow包
- 包括snow包在内的这四个流行包都使用了scatter/gather范式
- 多个进程/线程帮助我们进行并行计算，其中一个叫manager，其余叫worker
  - scatter: manager把计算任务分块，分发给worker
  - 块计算: worker在每个块上进行计算，将结果发送回manager
  - gather: manager把结果接收起来
- 在网站外链的例子中，不同worker负责for循环中不同i值对应的计算

例子：相互网页外链

- 并行代码

```
library(parallel)

doichunk <- function(ichunk){
  tot <- 0
  nr <- nrow(lnks)
  for(i in ichunk){
    tmp <- lnks[(i+1):nr, ] %*% lnks[i, ]
    tot <- tot + sum(tmp)
  }
  tot
}

mutoutpar <- function(cls, lnks){
  nr <- nrow(lnks)
  clusterExport(cls, "lnks")
  ichunks <- 1:(nr-1) #每个块对应一个值
  tots <- clusterApply(cls, ichunks, doichunk)
  Reduce(sum, tots) / (nr*(nr-1)/2)
}
```

- doichunk是将关于i的循环分解成多个块，让每个worker在分配给它的块上进行计算
- mutoutpar将整个过程包装起来，把i划分，然后在每个块上调用doichunk

例子：相互网页外链

- 并行代码

```
library(parallel)

doichunk <- function(ichunk){
  tot <- 0
  nr <- nrow(lnks)
  for(i in ichunk){
    tmp <- lnks[(i+1):nr, ] %*% lnks[i, ]
    tot <- tot + sum(tmp)
  }
  tot
}

mutoutpar <- function(cls, lnks){
  nr <- nrow(lnks)
  clusterExport(cls, "lnks")
  ichunks <- 1:(nr-1) #每个块对应一个值
  tots <- clusterApply(cls, ichunks, doichunk)
  Reduce(sum, tots) / (nr*(nr-1)/2)
}
```

- 用snow包的clusterExport来发送数据给worker，默认要求此数据是全局的
- 用snow包的clusterApply来指导worker对分配给它们的块进行计算，分配方式是轮询的(round robin)
- 调用R的核心函数Reduce将返回的结果结合起来(tots是一个列表)

例子：相互网页外链

- 并行代码

```
snowsim <- function(nr, nc, cls){
  lnks <<- matrix(sample(0:1, nr*nc, replace = TRUE), nrow = nr)
  system.time(mutoutpar(cls, lnks))
}

#在多核机器上设置一个有nworkers个worker的集群
initmc <- function(nworkers){
  makeCluster(nworkers)
}

#在制定的机器上设置集群
#每个机器一个worker
initcls <- function(workers){
  makeCluster(spec = workers)
}
```

- 创建lnks时，需要用到超赋值运算符<<-使得lnks是全局的

例子：相互网页外链

- 并行代码(两个内核，三个进程(1个manager, 2个worker))

```
#使用两个内核
cl2 <- makeCluster(2)
snowsim(2000, 2000, cl2)
```

- 运行结果(主要的计算都是由worker完成的，此运行时间只包括在流逝时间中)

```
用户   系统   流逝
0.35   0.53  25.57
```

- 速度提升了 33.21/25.57 = 1.299倍。这和理想的2倍不同，表明通信和其他开销也是一个主要因素
- 若不再用cl2，请用stopCluster(cl2)停止对应的snow集群

例子：相互网页外链

- 设置snow集群：snow集群是一个R的对象，不是物理实体，它包含了各种worker以及如何与其交互的信息
- 创建一个4节点的snow集群

```
> cls <- initmc(4)
> length(cls)
[1] 4
> cls
socket cluster with 4 nodes on host 'localhost'
```

```
> cls[[1]]
node of a socket cluster on host 'localhost' with pid 14200
> cls[[2]]
node of a socket cluster on host 'localhost' with pid 19296
> cls[[3]]
node of a socket cluster on host 'localhost' with pid 12688
> cls[[4]]
node of a socket cluster on host 'localhost' with pid 22960
```

- 创建指定机器名字的snow集群

```
> cl2 <- initcls(c("pc28", "pc29"))
```

  - 此操作在自己电脑上始终不会完成，因为自己电脑并没有链接到一个具有pc28、pc29的计算机集群

“我的程序为什么这么慢？”

- 一个常见的情景：一个分析师拿到了一台新的多核机器，带着激动的心情，写并行代码求解大规模问题，最终却发现运行速度比串行还慢
- 速度的障碍
  - 通信开销：数据会在进程间来回传输；不同进程去读取同一数据时，会相互阻塞
  - 负载平衡：若不注意任务分配方式，有的worker可能会分配较多的任务
- 缺乏性能和硬件结构知识
  - 多处理系统（比如，具有多个CPU的电脑），一般共享一个物理内存，因此也称为共享内存系统
  - 一个集群是多个可以独立运行的电脑构成，电脑之间联网进行通信

“我的程序为什么这么慢？”

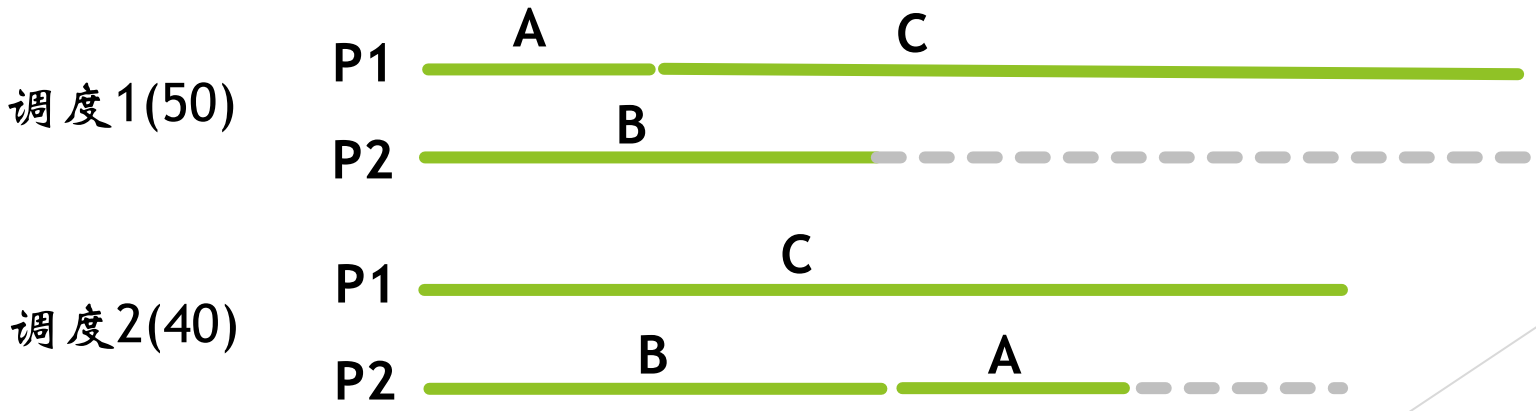
- 内存的基础知识
  - 矩阵的存储在R中使用**列主序**，比如z是一个6*5的矩阵，z[3,2]是在内存空间中的第6+3个
  - **高速缓存**是一个常用来解决内存读取缓慢问题的设备，是运行速度很快的内存块，位于处理器芯片附近
    - 处理器想读取的变量未在高速缓存中，则会去内存中寻找(消耗时间)---**高速缓存未命中**
  - **虚拟内存**：程序有一个变量x，其分配了地址888。但是在同一时间，还运行着其他程序，也可能给一个变量y分配地址888。一个标准解决方案是，让地址888变为“虚拟”的。
    - 第一个程序变量的物理地址为7660，程序仍会说地址为888，硬件执行程序时，将888翻译成7660。硬件有一个表来做查询。
    - 虚拟内存系统把内存划分为页，在某个时刻，只有部分程序的页在内存中，其余在硬盘上。若你的程序需要在硬盘中的页时，则需要从硬盘读入内存，这是在时间上极其昂贵的---**缺页错误**

“我的程序为什么这么慢？”

- 延迟和带宽
  - 通信信道的速度是由延迟和带宽来度量的
  - 延迟是一个信息从端到端的传输时间
  - 带宽是每秒钟能够往信道里发送的信息的数目
  - 假设延迟是l秒，带宽是b字节每秒，那么发送n个字节信息所花费的时间为 l + n/b

并行循环调度

- 循环调度的类型：
  - 静态调度：循环迭代是在执行开始之前就已经分配好
  - 动态调度：把循环迭代分配给进程的过程是在执行时进行的
- 例子：有两个进程P1和P2(即两个worker)；三个循环迭代A,B,C,其需要的时间分别为10，20，40.
- 考虑两种调度方式
  - 调度1：采用轮询的方式，A分配给P1，B分配给P2，C分配给P3
  - 调度2：考虑到A,B,C所耗费的时间，反顺序动态进行分配。
- 不同的调度方式会导致不同的负载不均衡



snow中的分块

- 我们利用clusterSplit进行分块

```
doichunk <- function(ichunk){
  tot <- 0
  nr <- nrow(lnks)
  for(i in ichunk){
    tmp <- lnks[(i+1):nr, ] %*% lnks[i, ]
    tot <- tot + sum(tmp)
  }
  tot
}


mutoutpar <- function(cls, lnks){
  nr <- nrow(lnks)
  clusterExport(cls, "lnks")
  ichunks <- clusterSplit(cls, 1:(nr-1)) #利用clusterSplit进行分块
  tots <- clusterApply(cls, ichunks, doichunk)
  Reduce(sum, tots) / (nr*(nr-1)/2)
}
```

- 假设有lnks有500行，我们有4个workers,clusterSplit将其**均分或大致均分**到每个worker上，即worker1有任务1~125 worker2有任务126~250， worker3有任务251~375，worker4有任务376~500

```
> clusterSplit(cls, 1:50)
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13

[[2]]
 [1] 14 15 16 17 18 19 20 21 22 23 24 25

[[3]]
 [1] 26 27 28 29 30 31 32 33 34 35 36 37

[[4]]
 [1] 38 39 40 41 42 43 44 45 46 47 48 49 50
```

snow中的分块

- 分块和不分块的比较（2个worker）

```
> cls <- makeCluster(2)
> #分块
> snowsim(2000, 2000, cls)
 用户   系统   流逝
 0.09   0.52 24.81
```

```
> #不分块
> snowsim(2000, 2000, cls)
 用户   系统   流逝
 0.53   0.85 27.20
```

snow中的分块

- 分块和不分块的比较（2个worker）

```
> cls <- makeCluster(2)
> #分块
> snowsim(2000, 2000, cls)
 用户   系统   流逝
 0.09   0.52 24.81
```

```
> #不分块
> snowsim(2000, 2000, cls)
 用户   系统   流逝
 0.53   0.85 27.20
```

例子：所有可能回归

- 线性回归中，n个观察值、p个预测变量，在变量选择的所有可能回归中，找到一个最为合适的模型。用统计的说法，就是一个变量选择的问题。在这个例子，将通过计算adjusted R^2 来进行选择。
- 并行化策略： 共有2^p种可能， 当p较大时，计算量就会相当的大
  - 给每个进程分配不同的预测变量集合，比如进程1对预测变量集合为{2,5}进行计算，进程2对预测变量集合为{8,9,12}进行计算
- 目标：在给定预测变量个数不超过k的条件下，通过adjusted R^2找到最合适的模型

例子：所有可能回归

- 代码(生成数据，将矩阵转化为列表)

```
####

#generate data
gendata <- function(n, p){
  x <<- matrix(rnorm(n*p), ncol=p)
  y <<- x %*% c(rep(0.5, p)) + rnorm(n)
}

#convert matrix rows (rc=1) or
# columns (rc=2) to list
matrixtolist <- function(rc, m){
  if(rc == 1){
    Map(function(rownum) m[rownum, ], 1:nrow(m))
  }else{
    Map(function(colnum) m[ , colnum], 1:ncol(m))
  }
}
```

- x,y是全局变量因此用超赋值符号<<

例子：所有可能回归

- 代码(生成所有可能的预测集合，以及计算某个预测集合对应的adjusted R2)

```
#generate a nonempty subset from {1,..,p}
# with size <= k; the returned value is
# an R list; each element in the R list
# is a set of predictors
genallcombs <- function(p, k){
  allcombs <- list()
  for(i in 1:k){
    tmp <- combn(1:p, i)
    allcombs <- c(allcombs, matrixtolist(tmp, rc=2))
  }
  allcombs
}

#calculate the adjusted R2 from
# a predictor set "onepset"
#returned is a k+1 dimensional vector
#e.g., k=4, (0.28, 1, 3, 0, 0)
do1pset <- function(onepset, x, y){
  slm <- summary(lm(y ~ x[ ,onepset]))
  n0s <- ncol(x) - length(onepset)
  c(slm$adj.r.squared, onepset, rep(0, n0s))
}
```

- combn(1:p, i)返回从1:p中不放回抽取i个元素的所有可能情况，是一个i行的矩阵(每一列对应一种可能情况,大家可以试试combn(1:5, 3))

例子：所有可能回归

- 代码：关于函数genallcombs的一个例子

```
> genallcombs(3,2)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 1 2

[[5]]
[1] 1 3

[[6]]
[1] 2 3
```

例子：所有可能回归

- 代码(返回预测变量集合索引从psetsstart到lasttask的adjusted R2情况)

```
#calculate adjusted R2s for one chunk
#psetsstart is the first index of this chunk
dochunk <- function(psetsstart, x, y, allcombs, chunksize){
  ncombs <- length(allcombs)
  lasttask <- min(psetsstart+chunksize-1, ncombs)
  t(sapply(allcombs[psetsstart:lasttask], do1pset, x, y))
}
```

- lasttask <- min(…)这一行命令是考虑所有可能组合数不能等分的情况，不能超过ncombs
- dochunk(3, x, y, allcombs, 4)即计算预测集合为allcombs[3], allcombs[4], allcombs[5], allcombs[6]所对应的adjusted R2
- sapply将结果是按列排列的，因此需要调用转置函数t

例子：所有可能回归

- 代码(主要的函数)

```
snowapr <- 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)

  allcombs <- genallcombs(p, k)
  ncombs <- length(allcombs)

  clusterExport(cls, "do1pset")

  #set the starting indices
  tasks <- if(!reverse) seq(1, ncombs, chunksize)
           else seq(ncombs, 1, -chunksize)

  if(!dyn){
    out <- clusterApply(cls, tasks, dochunk, x, y,
                        allcombs, chunksize)
  }else{
    out <- clusterApplyLB(cls, tasks, dochunk, x, y,
                          allcombs, chunksize)
  }
  Reduce(rbind, out)
}
```

例子：所有可能回归

- 代码：（主要函数中一个重要命令）

```
if(!dyn){
  out <- clusterApply(cls, tasks, dochunk, x, y,
                      allcombs, chunksize)
}else{
  out <- clusterApplyLB(cls, tasks, dochunk, x, y,
                        allcombs, chunksize)
}
```

- clusterApply指静态调度，若是两个worker1,worker2，那么worker1负责第1,3,5,...块，worker负责第2,4,6,...块
- clusterApplyLB指动态调度，worker1负责第1块，worker2负责第2块，剩下的调度我们提前就不知道了，依赖于程序执行中worker实际的完成情况。LB指负荷均衡(load balance)

例子：所有可能回归

- 代码(总结函数)

```
#the summarized function
snowtest <- function(cls, n, p, k, chunksize=1, dyn=F, rvrs=F){
  gendata(n, p)
  snowapr(cls, x, y, k, rvrs, dyn, chunksize)
}
```

- 一个简单结果

```
> c4 <- makeCluster(4)
> snowtest(c4, 100, 4, 2)
             [,1] [,2] [,3] [,4] [,5]
 [1,] 0.08985175    1    0    0    0
 [2,] 0.13924794    2    0    0    0
 [3,] 0.11805809    3    0    0    0
 [4,] 0.21869697    4    0    0    0
 [5,] 0.19920755    1    2    0    0
 [6,] 0.23318139    1    3    0    0
 [7,] 0.30849137    1    4    0    0
 [8,] 0.33275602    2    3    0    0
 [9,] 0.32308946    2    4    0    0
[10,] 0.34225709    3    4    0    0
```

例子：所有可能回归，计时实验

- 只调用一个核

```
> n <- 10000
> p <- 20
> k <- 3
> c1 <- makeCluster(1)
> system.time(snowtest(c1, n, p, k))
 用户   系统   流逝
 3.55 18.12 31.15
```

- 调用两个核

```
> c2 <- makeCluster(2)
> system.time(snowtest(c2, n, p, k))
 用户   系统   流逝
 3.03 17.78 25.77
```

- 利用动态调用

```
> system.time(snowtest(c2, n, p, k, dyn = TRUE))
 用户   系统   流逝
 3.03 17.84 20.92
```

- 尝试不同的chunksize

```
> system.time(snowtest(c2, n, p, k, dyn = TRUE, chunksize = 10))
用户 系统 流逝
0.37 1.86 5.72
> system.time(snowtest(c2, n, p, k, dyn = TRUE, chunksize = 20))
用户 系统 流逝
0.18 1.05 5.09
> system.time(snowtest(c2, n, p, k, dyn = TRUE, chunksize = 50))
用户 系统 流逝
0.11 0.37 4.93
```

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 $(X'X)^{-1}$ 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