分布式统计计算

2. 循环的并行

李舰

华东师范大学

2018-11-26

目 录

目 录

R 语言常用并行框架

- parallel 包
  - 从 R 2.14.0 开始内置 parallel 包。
  - 它基于 multicore 和 snow 包构建，提供了两个包的大部分功能。
- multicore 和 snow
  - multicore 包是早期支持多核运算的扩展包，只能在 Linux 下运行，后来被官方版本整合到 parallel 之后，就从 CRAN 上下架了。
  - snow 是早期 R 语言中很受欢迎的第三方包，提供了一个简单的框架用来实现并行计算，不仅可以在多核之间并行，还可以在网络集群中并行。
- foreach 包
  - foreach 是一种和 for 类似的循环框架，几乎可以通用，也可以很好地迁移。
  - 使用 %dopar% 运算符可以直接在并行后台（集群或者多核）中执行循环。
  - 可以通过 doMC、doParallel等包来注册并行的后台。

示例：相互网页外链

- 问题描述
  - 互联网上，任意一个网站都可能有链接连到另一个网站（包括自己），也都可能没有。
  - 对于任意两个网站，假如都链接到 A、B、C 这三个网站（各自可能还链接到其他网站），记这两个网站相互外链的数目为 3。
  - 所有网页两两之间相互外链的平均数是网页流量分析的重要指标，需要计算这个值。
- 外链矩阵
  - 记外链矩阵为 $links$，$links[i,j] = 1$ 表示存在网站 i 到 j 的外链，为 0 表示不存在外链。

$$links = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix}$$

串行代码

- 代码说明
  - 参数 links 传入一个外链矩阵。
  - 记数的变量 tot 初始值设为 0。
  - 通过 i 和 j 的循环进行两两比较。
  - 通过 k 循环判断每列中每个对应位置的值是否为 1。
- 代码示例

```
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)
}
```

串行代码运行时间

- 代码说明
  - 定义函数 sim 来随机产生外链矩阵。
  - 参数 nr 和 nc 分别表示矩阵的行数和列数。
  - 记录运行的时间。
- 代码示例

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

sim(500, 500)

##    user  system elapsed
##   10.34    0.01   10.42

sim(2000, 2000)

##    user  system elapsed
##  757.70    0.16  760.16
```

性能优化：改写成矩阵形式

- 代码说明
  - 定义一个新函数 mutoutser1。
  - k 层循环的部分其实对应向量的内积。
  - 用向量乘以矩阵可以代替 j 层循环。
  - 代码变得更简洁，但没那么易读。
  - 将循环改写成矩阵是 R 语言优化的一个重要思路。
- 代码示例

```
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
}
```

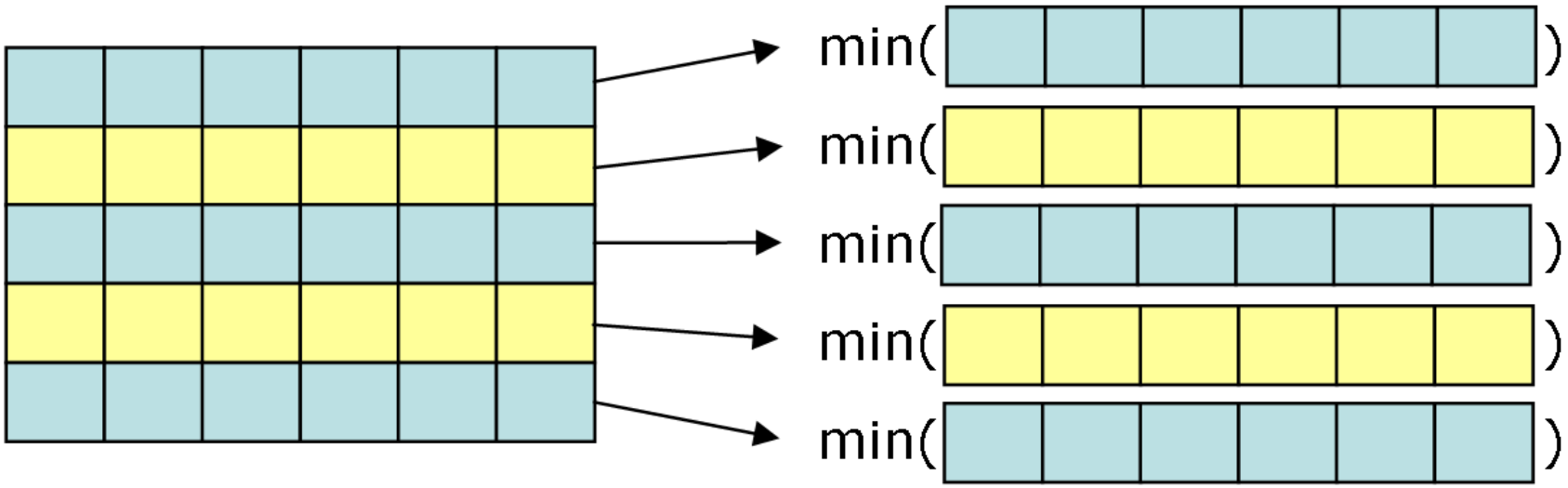
矩阵优化后运行时间

- 代码说明
  - 定义函数 sim1 来随机产生外链矩阵。
  - sim1 调用的是 mutoutser1 函数，其他不变。
  - 比较运行时间可以发现性能有很大的提升。
- 代码示例

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

sim1(500, 500)

##    user  system elapsed
##    0.91    0.15    1.06

sim1(2000, 2000)

##    user  system elapsed
##   79.51   12.16   92.03
```

目 录

apply 的本质，分而治之



apply 系列函数（I）

- apply(array, margin, function, ...)
  - 将一个函数作用于一个数组的某个维度，并返回一个降维后的数组。
  - 通过 margin 指定作用于哪个维度。
  - 对于二维的数据表（矩阵或者数据框）来说，margin 为 1 表示沿着第 1 个维度进行运算，也就是说对每行进行运算。margin 为 2 表示沿着第 2 个维度进行运算，也就是说对每列进行运算。
- lapply(list, function, ...)
  - 将函数作用于列表或向量的每个元素，并返回一个相同长度的列表，列表中的每个元素对应计算结果。
  - 很多时候可以代替循环。

apply 系列函数（II）

- sapply(list, function, ...)
  - 将函数作用于列表或向量的每个元素，并返回一个向量、矩阵或者列表。
  - 如果 simplify 设为 FALSE，则等价于 `lapply`。
  - 默认 simplify 为 TRUE，会根据函数值自动判断返回的数据类型，如果是数值则返回向量，如果是等长的量则返回矩阵，否则返回列表。
- tapply(array, indicies, function, ..., simplify)
  - 将一个函数作用于一个不规则数组（ragged array）中的每个单元。
  - 对某个数组按照位置分好组，然后对每组进行函数运算。
  - 可以按照因子进行分组。

目 录

继续优化：并行

- 代码说明
  - 使用 parallel 包里的 clusterApply 函数实现并行。
  - mutoutpar 是并行框架函数，调用算法函数 doichunk。
- 代码示例

```
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)
   tots <- clusterApply(cls,ichunks,doichunk)
   Reduce(sum,tots) / nr
}
```

矩阵加并行优化后运行时间

- 代码说明
  - parallel 包中的 makeCluster 函数用来初始化并行后台。
  - 通过双核计算后可以发现性能有提升，但远远没有达到原来的两倍。
- 代码示例

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

library(parallel)
cl2 <- initmc(2)
snowsim(2000, 2000, cl2)

##    user  system elapsed
##    0.50    0.16   66.64
```

性能优化：使用 foreach

- 代码说明
  - 使用 foreach 包里的 foreach 函数将 for 循环进行改写，通过 %dopar% 运算符实现并行。
  - 其他代码保持串行的原样。
- 代码示例

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

foreach 优化后运行时间

- 代码说明
  - 借助 doParallel 包来自动生成供 foreach 运行的后台。
  - 从结果可以发现相比于串行的 for 循环是有提升的。
  - foreach 最大优势是编程容易，但是也可以优化设计后整合矩阵运算提升效率。
- 代码示例

```
simfe <- function(nr, nc, ncores = 2) {
  require(doParallel)
  cls <- makePSOCKcluster(ncores)
  registerDoParallel(cls)
  lnk <<- matrix(sample(0:1,(nr*nc),replace=TRUE),
    nrow=nr)
  print(system.time(mutoutfe(lnk)))
  stopCluster(cls)
}
simfe(500, 500)

##    user  system elapsed
##    0.34    0.06    6.99
```

目 录

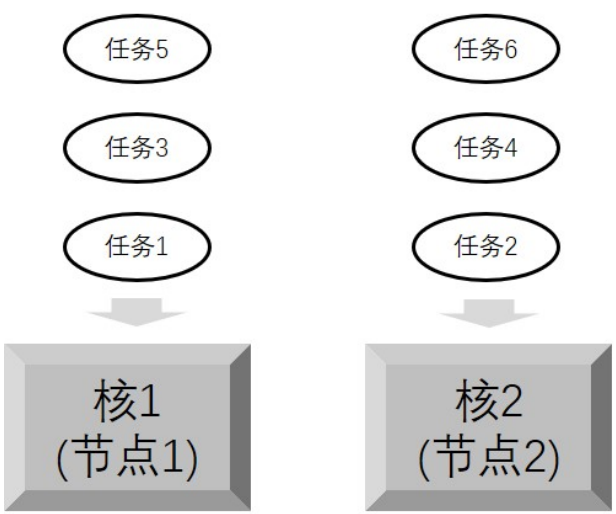
目 录

循环并行简介

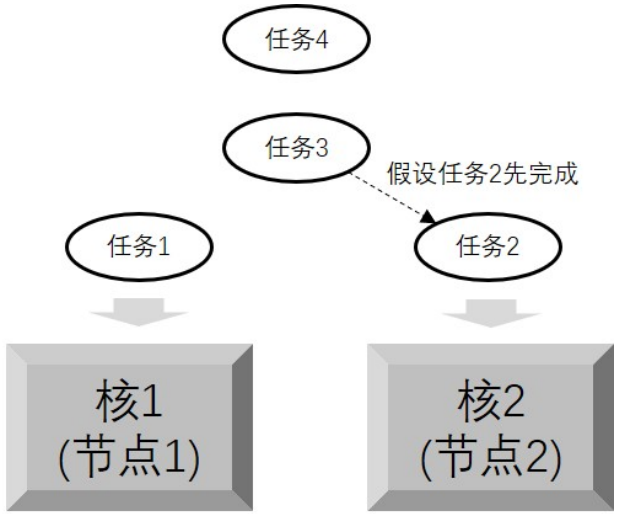
- 易并行
  - 易并行（Embarrassing）指的是简单的并行，这类问题一般实在太简单，不涉及任何智力上的挑战。
  - 循环的并行是典型的易并行问题。
  - 循环的并行原理非常简单，但是要注意工程上的调度问题。
- 并行循环调度
  - 静态调度：循环迭代是在执行开始之前分配给循环迭代的进程的。
  - 动态调度：把循环迭代分配给进程的过程是在执行时进行的。每当一个进程结束一个循环迭代时，它拾起一个（或一组）新的循环迭代来继续工作。
  - 分块：将一组而非一个循环迭代分配给一个进程。
  - 反向调度：有时候一个迭代的执行时间会随着循环的进行越来越长，把迭代的顺序反向会更加高效。

调度示例

- 注意事项
  - 如果每一个工作任务代表着一组迭代任务，则为分块调度。
  - 如果按照反向排序，则为反向调度。假设有 3 个任务 A、B、C 的执行时间分别为 10、20、30，如果正向调度，总时间为 40，如果反向调度，总时间为 30。



静态调度



动态调度

示例：所有可能回归

- 问题描述
  - 假设我们有 n 个观测值，p 个预测变量，进行线性回归分析。
  - 回归分析有一个关键的步骤在于变量的筛选，变量太多的时候通常会存在多重共线性问题，我们可以用逐步回归的方法基于 AIC 筛选变量。
  - 最好的方法应该是基于可决系数 R^2 来筛选变量，这就需要我们遍历 p 个变量的每一个可能的子集，一共存在 2^p 种可能的模型。
  - 我们穷举每一种可能，然后分别建模，最终选择 R^2 最高的模型，需要通过并行计算来解决。
- 实现的两种思路
  - 在算法层面并行，对每一种预测变量的组合，调用所有进程一起工作，这种方式的算法很复杂，比如需要计算矩阵的逆（或者 QR 分解这样的等价运算），并不是易并行的问题。
  - 对每个进程分配不同的变量组合，这样每个进程每次做一个回归分析，这是一个循环并行的问题。

目 录

计算任务列表

- 代码说明
  - 函数 genallcombs 用来生成预测变量的所有组合，返回一个列表对象，其中每个元素代表一种组合。
  - 参数 p 代表所有变量的数目，参数 k 代表要选用变量的数目。
  - 函数 matrixtolist 用来把函数 combn 的结果转成列表。
- 代码示例

```
genallcombs <- function(p,k) {
  allcombs <- list()
  for (i in 1:k) {
    tmp <- combn(1:p,i)
    allcombs <- c(allcombs,matrixtolist(tmp,rc=2))
  }
  allcombs
}
matrixtolist <- function(rc,m) {
  if (rc == 1) {
    Map(function(rownum) m[rownum,],1:nrow(m))
  } else Map(function(colnum) m[,colnum],1:ncol(m))
}
```

核心程序：关键子函数

- 代码说明
  - 函数 do1pset 用来做线性回归分析，参数 x 和 y 分别代表自变量和因变量，参数 $onepset$ 代表筛选后的变量，该函数输出 R^2 和选择的变量编号。
  - 函数 dochunk 用来处理任务组合的分块，将任务根据块的数目分组，并对每组数据运行 do1pset 函数。
- 代码示例

```
dochunk <- function(psetstart,x,y,allcombs,chunk) {
  ncombs <- length(allcombs)
  lasttask <- min(psetstart+chunk-1,ncombs)
 t(sapply(allcombs[psetstart:lasttask],do1pset,x,y))
}

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))
}
```

核心程序：主体函数代码

- 代码说明
  - 函数 snowapr 调用 dochunk 和 dochunk，实现并行计算。
- 代码示例

```
snowapr <- function(cls, x, y, k, reverse = FALSE,
  dyn = FALSE, chunk = 1) {
  p <- ncol(x)
  allcombs <- genallcombs(p,k)
  ncombs <- length(allcombs)
  clusterExport(cls,"do1pset")
  tasks <- if (!reverse) seq(1,ncombs,chunk) else
    seq(ncombs,1,-chunk)
  if (!dyn) {
    out <- clusterApply(cls, tasks, dochunk, x, y,
           allcombs, chunk)
  } else {
   out <- clusterApplyLB(cls, tasks, dochunk, x, y,
           allcombs, chunk)
  }
  Reduce(rbind,out)
}
```

核心程序：主体函数简介

- 函数简介
  - 函数 snowapr 依赖 parallel 包，实现了整个工作的并行计算，并返回所有结果的矩阵（每种变量组合的 R^2）。
  - 函数 clusterExport 用来把变量发布到集群中。
  - 变量 tasks 用来设定任务的编号，并考虑到调度块的规模，基于参数 chunk 计算每块的任务编号。
  - clusterApply 是默认的并行执行函数。clusterApplyLB 考虑了负载均衡，可以实现动态调度。
- 参数简介
  - 参数 x、y、k 分别代表自变量数据、因变量数据、筛选变量的数目。
  - 参数 cls 代表 SNOW 的后台集群。
  - 参数 reverse 表示是否进行反向调度。
  - 参数 dyn 表示是否进行动态调度。
  - 参数 chunk 表示调度块的规模。

运行和测试

- 代码说明
  - 函数 gendata 用来随机生成自变量矩阵和因变量向量。
  - 函数 test 用来调用并行函数并记录时间。
  - parallel 包中的 makeCluster 函数用来建立并行的后台集群，我们分别尝试使用双核和 4 核的效果。
- 代码示例

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

test <- function(cls,n,p,k,chunk=1,dyn=F,rvrs=F) {
  gendata(n,p)
  system.time(snowapr(cls,x,y,k,rvrs,dyn,chunk))
}

library(parallel)
cl2 <- makeCluster(2)
cl4 <- makeCluster(4)
```

运行结果

```
test(cl2, 10000, 20, 3, dyn = FALSE)

##    user  system elapsed
##    5.39    4.91   25.14

test(cl2, 10000, 20, 3, dyn = TRUE)

##    user  system elapsed
##    5.64    4.36   20.39

test(cl2, 10000, 20, 3, dyn = TRUE, chunk = 50)

##    user  system elapsed
##    0.07    0.06   12.61

test(cl4, 10000, 20, 3, dyn = TRUE, chunk = 50)

##    user  system elapsed
##    0.11    0.12    8.56
```

算法的改进

- 原理
  - 深入到回归模型，可以知道回归系数估计值的计算方式：

$$\hat{\beta} = (X'X)^{-1}X'Y$$

  - 其中 X 是自变量的矩阵，Y 是因变量的向量。
  - 具体计算中，我们可以使用 QR 分解之类的方法来替代矩阵求逆，通过一些基础的矩阵运算得到估计值。
  - 我们可以发现，无论我们选择那些变量建模，都会用到 $X'X$ 的子集，在我们之前的并行算法中，直接调用 `lm` 函数，实际上进行了大量的重复计算，如果我们把 $X'X$ 存为中间值，用自己编写的函数代替 `lm`，可以很好地提升性能。
- 实现
  - 改写 `lm` 函数，如果模型包含常数项，在自变量矩阵中添加一列全为 1 的向量。
  - 我们计算一次 $X'X$ 的值，将其发布到整个集群全局的环境中，供每个节点调用，各节点可以根据自己的变量从中取子集，带入到后续的计算中。

目 录

multicore 简介

- Unix 族和 fork
  - 类 Unix 系统（包括 Linux）都有被称为 `fork` 的系统调用，即写应用的程序员可以调用系统的一个函数作为服务。
  - 这种方式相当于自动复制多个 R 的实例，如果我们将核心数设置为 4，意味着系统会自动新开启 4 个 R 实例，这样内存中一共存在 5 个实例，父实例的进程会自动休眠，等子进程运行结束。
  - 通过 `fork` 方式复制的进程和父进程一模一样，并共享数据（存取内存中相同的物理地址）。
  - 在 SNOW 的方式下，把数据传输到节点需要时间和资源（即使在单机也需要），但是 `fork` 的方式可以在本机共享数据，省了传输的过程，会更加高效。
- multicore 的应用
  - 原始的 `multicore` 包基于 `fork` 的机制实现并行，在新的 `parallel` 包中通过 `mclapply` 函数来实现。
  - Windows 系统下并不支持 `fork` 的机制，所以只能在类 Unix 系统下尝试该功能。

Thank you!

李舰 Email: jian.li@188.com