分布式统计计算

3. 共享内存机制

李舰

华东师范大学

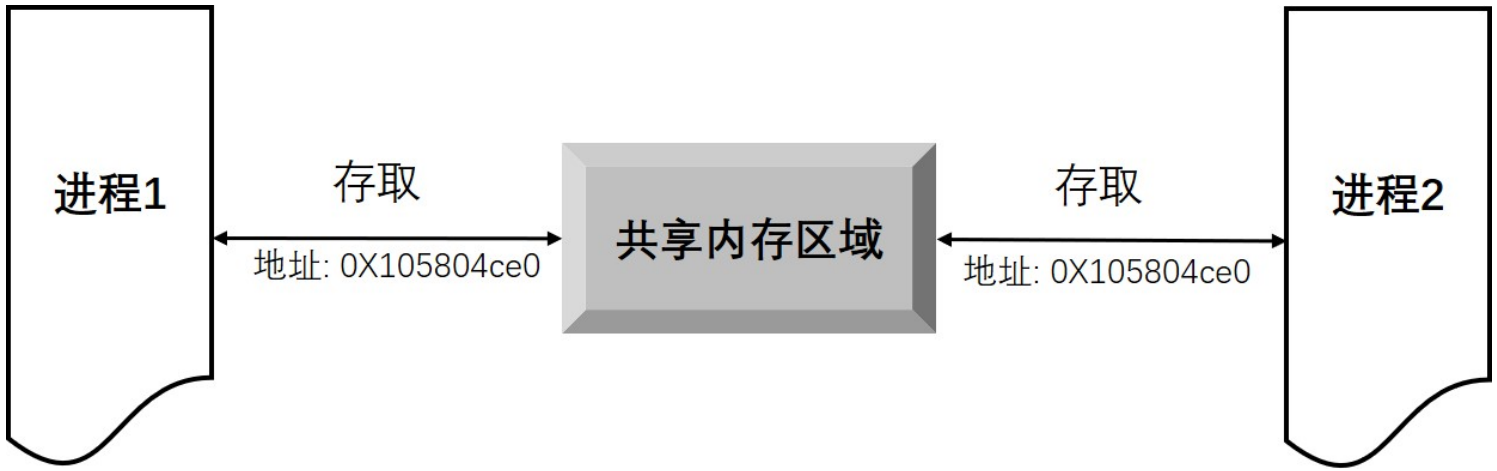
2018-12-03

目 录

目 录

共享内存范式

- **机制和范式**
  - 多个并行的进程通过存取机器中他们所共同使用的内存单元来互相通信，称为共享内存范式。
  - 最常见的共享内存硬件类型是多核机器。
  - 另一个共享内存硬件类型是加速芯片，特别是图形处理单元（GPU）。
  - 多线程编程中每个进程的所有线程都是共享内存的。
  - 共享内存的进程（线程）通过相同的内存地址访问相同数据。



R 语言共享内存

- Rdsm 包简介
  - Norm Matloff 开发，提供了一个线程型的编程环境。
  - 依赖 parallel 包和 bigmemory 包。
  - 在 Rdsm 中每个进程都是一个单独、独立的 R 实例，使用的时候，进程必须运行在同一台机器上，通过物理共享内存来共享变量，适合作为入门的示例。
  - 目前只能运行于类 Unix 的机器。
- bigmemory 简介
  - 在现代操作系统，一块内存都可以被任何一个包含键代码的进程所共享，bigmemory 包可以实现这种机制。
  - 在 bigmemory 中，共享变量必须使用矩阵的形式（标量可以看成是 1×1 的矩阵），共享矩阵的类是 big.matrix，Rdsm 继承了这些特性。
  - 假设某个 big.matrix 对象 m，在 R 中运行 m 输出共享内存对象的地址，运行 m[,] 输出矩阵的值。
  - bigmemory 还适用于存储在共享磁盘文件上的共享变量，因此可以用于内存外运算和分布式系统。

矩阵乘法共享函数

- 代码说明
  - 定义函数 mmulthread 运行于每个线程。
  - 参数 u 和 v 代表进行矩阵乘法的左右两个矩阵，参数 w 是结果矩阵，在运算的过程中其值会改变。
  - myinfo 是 Rdsm 包内置的一个 R 列表对象，其 nwrkrs 元素代表线程总数，id 表示当前线程 ID。
  - parallel 包中的 splitIndices 函数用来对线程分配块，可以通过线程 ID 从共享内存中取得当前计算的数据。
  - 最后输出的 0 并无实际意义，只是为了避免比较耗资源的 return 函数输出的返回值。
- 代码示例

```
mmulthread <- function(u,v,w) {
  require(parallel)
  myidxs <- splitIndices(nrow(u),
      myinfo$nwrkrs)[[myinfo$id]]
  w[myidxs,] <- u[myidxs,] %*% v[,]
  0
}
```

矩阵乘法测试函数

- 代码说明
  - 定于基于集群后台（多核）cls 的测试函数 test。
  - 函数 mgrinit 用来初始化集群。
  - 函数 mgrmakevar 用来设置共享变量的名称和维度（2 维）。
  - 函数 clusterExport 把自定义函数发布到集群的节点。
  - 函数 clusterEvalQ 在集群上运行一个表达式。
- 代码示例

```
test <- function(cls) {
  mgrinit(cls)
  mgrmakevar(cls,"a",6,2)
  mgrmakevar(cls,"b",2,6)
  mgrmakevar(cls,"c",6,6)
  a[,] <- 1:12
  b[,] <- rep(1,12)
  clusterExport(cls,"mmulthread")
  clusterEvalQ(cls,mmulthread(a,b,c))
  print(c[,])
}
```

矩阵乘法测试结果

- 代码说明
  - 运行前先加载依赖包 parallel 和 Rdsm。
  - 首先建立 2 核集群 c2，注意该集群可以用于 Snow 模式的消息传递并行机制，也可以用于共享内存的并行机制。
  - 调用 test 函数时将之前定义的 mmulthread 函数应用于每个线程，从共享内存中获取数据，因此省掉了数据复制。
- 代码示例

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

基于 Snow 的矩阵乘法

- 代码说明
  - 定于函数 snowmmul 执行于整个集群。
  - 参数 cls 代表集群后台，参数 u 和 v 代表相乘的左右两个矩阵。
  - 在函数内部定义 mmulchunk 函数用来执行在各个节点。
  - 最后通过 Reduce 函数将每个节点的运行结果（向量）拼接成一个矩阵。
- 代码示例

```
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)
}
```

测试两种并行方式的性能

- 代码说明
  - 定义函数 testcmp 同时执行两种并行方式。
  - 参数 n 代表方阵的行列数目。
  - 在同样的集群 cls 上分别运行 mmulthread 和 snowmmul。
- 代码示例

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

测试结果

- 代码说明
  - 针对不同大小的矩阵进行测试。
- 代码示例

```
testcmp(c2, 500)

##    user  system elapsed
##   0.001   0.000   0.079
##    user  system elapsed
##   0.029   0.006   0.122

testcmp(c2, 1000)

##    user  system elapsed
##    0.00    0.00    0.59
##    user  system elapsed
##   0.117   0.025   0.802

testcmp(c2, 2000)

##    user  system elapsed
##   0.000   0.000   4.496
##    user  system elapsed
##   0.530   0.108   4.847
```

目 录

锁（Lock）

- 锁机制
  - 在并行程序中可能存在各节点都要读写某些资源的情况，相关的代码区域称为临界区。
  - 我们需要一种机制，可以限制在同一个时刻只有一个线程可以存取临界区，这被称为互斥。
  - 一种常见的机制是锁变量（Lock Variable）或者互斥量（Mutex）。
  - 在 `Rdsm` 包中使用 `mgrmakelock` 函数创建锁变量，用 `rdsmlock` 和 `rdsmunlock` 实现上锁与解锁。
- 示例
  - 一个加法运算的循环，对某个变量 `tot` 的值每次加 1，求最终的数值。
  - 在每个节点中，该循环会先读取 `tot` 的当前值，进行加 1 运算后将新的数值写回 `tot` 。
  - 存在两个节点取了相同的中间值，并先后写回相同数值的情况，因此最终的结果会存在错误。

累加示例：未上锁

- 代码说明
  - 定义函数 s 实现累加的操作，然后使用共享内存的机制实现并行，2 个线程各 1000 次累加后得到 1058。
- 代码示例

```
s <- function(n) {
    for (i in 1:n) {
        tot[1, 1] <- tot[1, 1] + 1
    }
}

clusterExport(c2, "s")
mgrinit(c2)

mgrmakevar(c2, "tot", 1, 1)
tot[1, 1] <- 0
clusterEvalQ(c2, s(1000))

tot[1, 1]

## [1] 1058
```

累加示例：上锁

- 代码说明
  - 定义函数 s1，实现锁机制，创建锁变量 totlock，插入到临界区，最终可以得到正确结果 2000。
- 代码示例

```
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))

tot[1, 1]

## [1] 2000
```

屏障（Barrier）

- 屏障机制
  - 如果我们需要一个线程来进行一个特殊的操作，并且需要让其他线程等待这个操作结束，可以通过屏障机制来实现。
  - 在 Rdsm 包中提供了 barr 函数，当一个线程调用它时，该线程会被阻塞，直到所有线程都调用了这个函数，然后它们都继续执行下一行代码。
  - 本质上屏障机制都是通过锁来实现的，但是我们可以直接使用该机制而无需深入设置锁的细节。
- 示例
  - 假设我们有一个长度为 10 的序列（向量）x，计算窗口长度为 k（本例为 2）的滑动平均，输出最大的滑动平均值（也称为最大脉冲值）以及该窗口的起始位置。
  - 我们使用一个长度为 9 的向量 mas 作为暂存空间，用一个长度为 2 的向量 rslts 保存结果（第 1 个元素为起始位置，第 2 个元素为最大脉冲值）。
  - 每一段脉冲的计算是独立的，因此很容易并行，使用 zoo 包中的 rollmean 函数来计算滑动平均，都结束后计算最大值。

最大脉冲示例：设置屏障

- 代码说明
  - 在 rollmean 计算结束后设置屏障，然后计算最大值。
  - getidxs 函数是对 splitIndices 结合 myinfo 的一个封装。
- 代码示例

```
maxburst <- function(x, k, mas, rslts) {
  require(Rdsm)
  require(zoo)
  n <- length(x)
  myidxs <- getidxs(n-k+1)
  myfirst <- myidxs[1]
  mylast <- myidxs[length(myidxs)]
  mas[1, myfirst:mylast] <-
      rollmean(x[myfirst:(mylast+k-1)], k)
  barr()
  if (myinfo$id == 1) {
    rslts[1, 1] <- which.max(mas[ , ])
    rslts[1, 2] <- mas[1, rslts[1, 1]]
  }
}
```

最大脉冲示例：运行

- 代码说明
  - 基于共享内存的机制实现并行计算，输出的结果向量显示起始位置为 9 的脉冲最大，其值为 16。
- 代码示例

```
test <- function(cls) {
  require(Rdsm)
  mgrinit(cls)
  mgrmakevar(cls, "mas", 1, 9)
  mgrmakevar(cls, "rslts", 1, 2)
  x <<- c(5, 7, 6, 20, 4, 14, 11, 12, 15, 17)
  clusterExport(cls, "maxburst")
  clusterExport(cls, "x")
  clusterEvalQ(cls, maxburst(x, 2, mas, rslts))
  print(rslts[, ])
}

test(c2)

## [1]  9 16
```

目 录

目 录

邻接矩阵简介

- 邻接矩阵和边矩阵
  - 邻接矩阵 A 的元素 a_{ij} 表示是否存在从 i 到 j 的关系。
  - 我们可以将其转化为边矩阵 B，每一行代表序号 b_{i1} 的元素到序号 b_{i2} 的元素存在关系。

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \quad \Rightarrow \quad B = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 2 & 4 \\ 3 & 2 \\ 3 & 4 \\ 4 & 1 \\ 4 & 2 \\ 4 & 3 \end{bmatrix}$$

- 并行计算
  - 假定一个邻接矩阵 `adj`，我们将其转化成边矩阵 `lnks`，但事先未知其行数，可以预估为最坏情况 `nrow(adj)^2`。还定义 `counts` 为每个线程找到的边的数目，从而可以删除边矩阵中多余的行。

定义节点的函数

- 代码说明
  - 定义并调用 convert1row 函数针对每一行进行转换。
- 代码示例（I）

```
convert1row <- function(rownum, colswith1s) {
  if (is.null(colswith1s)) return(NULL)
  cbind(rownum, colswith1s)
}

findlinks <- function(adj,lnks,counts) {
  require(parallel)
  nr <- nrow(adj)
  myidxs <- getidxs(nr)
  myout <- apply(adj[myidxs,],1,
      function(rw) which(rw==1))
  tmp <- matrix(nrow=0, ncol=2)
  my1strow <- myidxs[1]
  for (idx in myidxs) {
    tmp <- rbind(tmp, convert1row(idx,
              myout[[idx - my1strow + 1]]))
  }
```

定义节点的函数

- 代码说明
  - 对每个节点处理的行数进行统计，汇总后存入 counts 变量。
  - 利用屏障的机制，等每个节点的汇总都完成后，利用某个进程（比如第 1 个）进行加总。
  - 然后将各节点计算得到的边矩阵写入输出结果的相应位置。
- 代码示例（II）

```
  nmyedges <- Reduce(sum,lapply(myout,length))
  me <- myinfo$id
  counts[1,me] <- nmyedges
  barr()
  if (me == 1) {
    counts[1,] <- cumsum(counts[1,])
  }
  barr()
  mystart <- if(me == 1) 1 else counts[1, me-1] + 1
  myend <- mystart + nmyedges - 1
  lnks[mystart:myend, ] <- tmp
  0
}
```

测试运行

- 代码说明
  - 将自定义的函数 findlinks 和 convert1row 都通过 clusterExport 发布到集群 cls 中。
  - 基于共享内存机制实现并行计算。
- 代码示例

```
test <- function(cls) {
  mgrinit(cls)
  mgrmakevar(cls,"x",6,6)
  mgrmakevar(cls,"lnks",36,2)
  mgrmakevar(cls,"counts",1,length(cls))
  x[,] <- matrix(sample(0:1,36,replace=T),ncol=6)
  clusterExport(cls,"findlinks")
  clusterExport(cls,"convert1row")
  clusterEvalQ(cls,findlinks(x,lnks,counts))
  print(lnks[1:counts[1,length(cls)],])
}

test(c2)
```

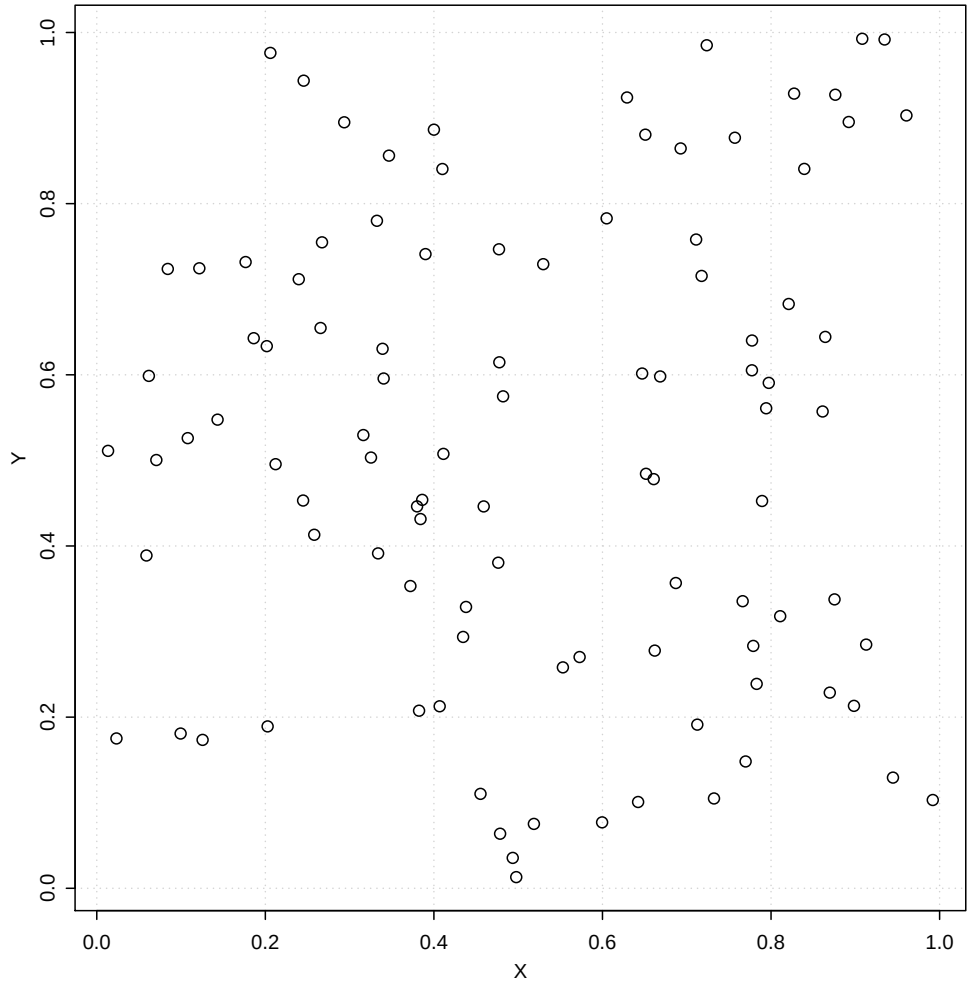
串行函数

- 代码说明
  - 使用串行函数对行进行循环也可以计算，可以用来对比运行时间。
- 代码示例

```
getlinksnonpar <- function(a,lnks) {
  nr <- nrow(a)
  myout <- apply(a[,],1,function(rw) which(rw==1))
  nmyedges <- Reduce(sum,lapply(myout,length))
  lnksidx <- 1
  for (idx in 1:nr) {
    jdx <- idx
    myoj <- myout[[jdx]]
    endwrite <- lnksidx + length(myoj) - 1
    if (!is.null(myoj)) {
      lnks[lnksidx:endwrite, ] <- cbind(idx, myoj)
    }
    lnksidx <- endwrite + 1
  }
  0
}
```

目 录

k-means 聚类的算法原理



k-means 并行

- 并行思路
  - k-means 算法的迭代过程需要依赖上一次的结果，所以这个过程无法并行。
  - 每一步迭代中，计算所有点和中心点距离的过程可以并行，通过锁和屏障机制控制在一个迭代内。
  - 每次迭代结束后通过某个线程（比如第 1 个）线程设置新的中心点，然后进入下一次迭代。
- R 的实现
  - 假设 `x` 为数据矩阵，`k` 为类的数目，`ni` 为迭代次数，`cntrds` 是中心矩阵，`cinit` 是中心初始值，`sums` 是暂存矩阵，`sums[j, ]` 包含了对第 `j` 类的计数的求和，`lck` 是共享的锁变量。
  - 使用 `pdist` 包的 `pdist` 函数来计算距离，其结果是一个 `S4` 的对象，需要通过 `@` 符号来提取元素信息。
  - 使用 `barr` 实现屏障，使用 `realrdsmlock` 和 `realrdsmunlock` 实现锁机制。

k-means 并行函数（I）

```
kmeans<-function(x,k,ni,cntrds,sums,lck,cinit=NULL) {
   require(parallel)
   require(pdist)
   nx <- nrow(x)
   myidxs <- getidxs(nx)
   myx <- x[myidxs,]
   if (is.null(cinit)) {
      if (myinfo$id == 1)
         cntrds[,] <- x[sample(1:nx,k,replace=F),]
      barr()
   } else cntrds[,] <- cinit
   mysum <- function(idxs,myx) {
      c(length(idxs),colSums(myx[idxs,,drop=F]))
   }
   for (i in 1:ni) {
      if (myinfo$id == 1) {
         sums[] <- 0
      }
      barr()
```

k-means 并行函数（II）

```
        dsts <- matrix(pdist(myx,cntrds[,])@dist,
                ncol=nrow(myx))
        nrst <- apply(dsts,2,which.min)
        tmp <- tapply(1:nrow(myx),nrst,mysum,myx)
        realrdsmlock(lck)
        for (j in as.integer(names(tmp))) {
           sums[j,] <- sums[j,] + tmp[[j]]
        }
        realrdsmunlock(lck)
        barr()
        if (myinfo$id == 1) {
           for (j in 1:k) {
             if (sums[j,1] > 0) {
                cntrds[j,] <- sums[j,-1] / sums[j,1]
              } else cntrds[j] <<- x[sample(1:nx,1),]
           }
        }
    }
    0
}
```

Thank you!

李舰 Email: jian.li@188.com