分布式统计计算

5. 消息传递机制

李舰

华东师范大学

2018-12-17

目 录

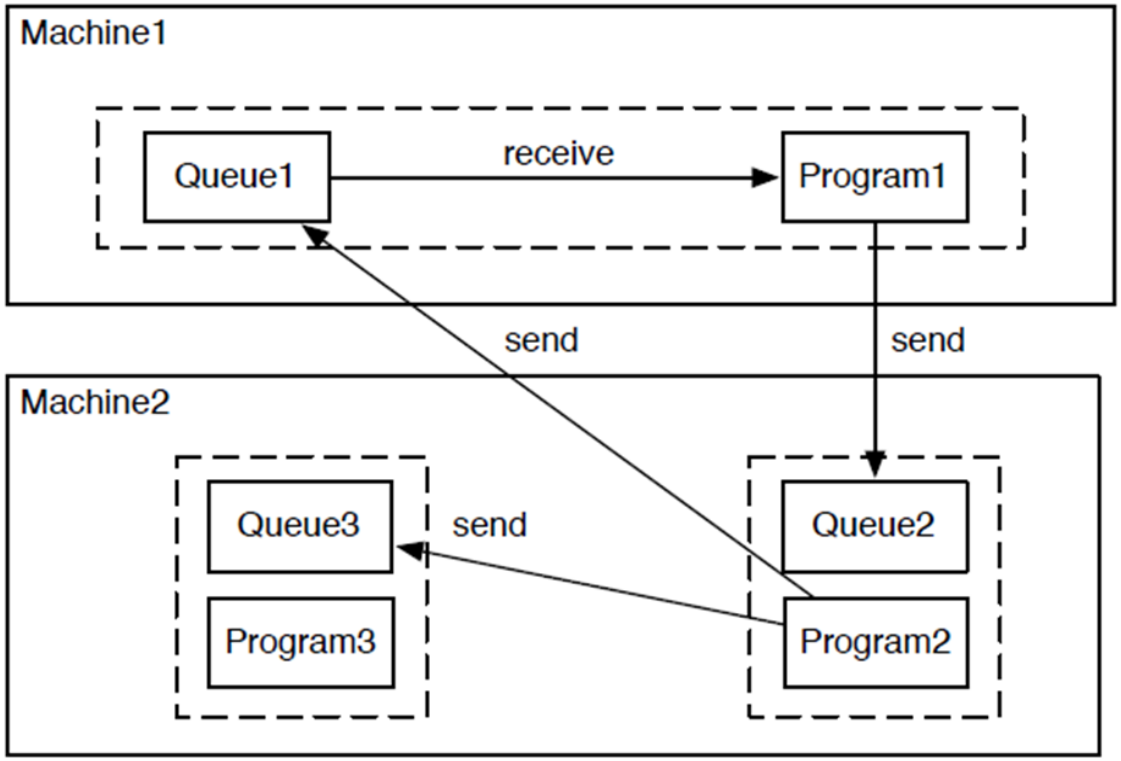
目 录

消息传递概述

- 常用术语
  - 主机，也称为 Manager 或者 Master。从机，也称为 Worker 或者 Slave。
  - 在之前介绍的 snow 中，从机只和主机进行通信，如果我们允许从机之间也可以互相通信的话，这种情况一般称为“消息传递范式”。
- 实现机制
  - 消息传递是一个软件（算法）上的概念，因此并不要求硬件平台具有特殊的结构。
  - 消息传递一般被认为运行在一个集群上，比如由独立机器构成的网络，每台机器有其自己的处理器和内存。
  - 最流行的消息传递的 C 函数库是消息传递接口（Message Passing Interface，简称 MPI），有不同的系统实现版本，于 1991 年在奥地利发起，1992 年形成了一套标准，1997 年进行了重大扩充。

MPI 简介

- 什么是 MPI
  - 定义了核心库的语法和语义，该库可以被 C 和 FORTRAN 调用从而构成可移植的消息传递程序。
- MPI 实现机制



安装 MPI 环境

- MPI 常用版本
  - OpenMPI: https://www.open-mpi.org/, MPI-2 标准的一个开源实现。
  - MPICH2: http://www.mpich.org/, 一个跨平台的 MPI 项目，支持最新的 MPI-2 接口标准。
  - Microsoft MPI: 微软提供的 Windows 平台下的 MPI 环境。
- Windows 下安装
  - 进入微软下载页面 https://msdn.microsoft.com/en-us/library/windows/desktop/bb524831%28v=vs.85%29.aspx 下载 MSMpiSetup.exe 安装文件。
  - 自动安装的话会默认安装到 C:\Program Files\Microsoft MPI 文件夹，并新建环境变量。
  - 也可以进行编译安装，能提高性能，但需要自行配置环境，并手动添加 MPI_HOME 环境变量。

目 录

Rmpi 安装

- 什么是 Rmpi
  - 西安大略大学的教授 Hao Yu 开发的 R 程序包，提供了 R 的 MPI 接口，同时也增加了一些 R 特有的函数。
  - 主页：http://www.stats.uwo.ca/faculty/yu/Rmpi/。
- 安装 Rmpi 包
  - 以 Windows 为例，首先安装 Microsoft MPI 环境。
  - 然后在 R 的界面下安装：

```
install.packages("Rmpi")
```

- 运行
  - 进入某文件夹，将 Rmpi 包中的 Rprofile 文件复制到该文件夹，并通过命令行重命名为 .Rprofile：

```
REN Rprofile .Rprofile
```

  - 然后在命令行利用管理员权限执行如下语句：

```
mpiexec  -n 5 Rterm.exe --no-save -q
```

通过 RStudio 打开 MPI 模式

- 新建快捷方式
  - 当前版本的 Rmpi 支持从 RStudio 中启动 MPI 模式，在桌面点击右键，选择“新建”-> “快捷方式”。
  - 把 "C:\Program Files\Microsoft MPI\Bin\mpiexec.exe" -n 1 "C:\Program Files\RStudio\bin\rstudio.exe" 复制粘贴到输入框。注意，连同引号一起复制，如果不是默认安装路径，修改成自己的路径。

运行 Rmpi

- 运行机制
  - 在 MPI 模式下，加载 Rmpi 包可以开启环境。
  - 函数 mpi.comm.rank 可以用来识别各自的 ID。
  - 结束后调用 mpi.close.Rslaves 和 mpi.exit 函数来退出。
- 示例代码

```
library(Rmpi)
mpi.remote.exec(paste("I am", mpi.comm.rank(), "of",
    mpi.comm.size()))

## $slave1
## [1] "I am 1 of 5"
##
## $slave2
## [1] "I am 2 of 5"
##
## $slave3
## [1] "I am 3 of 5"
##
## $slave4
## [1] "I am 4 of 5"
```

一个简单的例子

- 回归分析的例子
  - 假设进行回归分析，将数据随机地分成 4 部分，然后分别交给 4 台机器（本例中是 4 个核）计算，这样可以估计出 4 套系数。
  - 随机模拟一批数据，只包含变量 x 和 y，一共 1 万行。
  - 简单随机抽样将其分到四个组，变量 sampleid 记录了每条记录分配的组号。
  - 在这个简单的例子中，每个从机的任务是一样的，没有先后之分。因此以使用一个最简单的消息传递方式“bcast”，也就是广播，将信息同时“广播”到所有的从机。
- 示例代码

```
lmdata <- data.frame(y = rnorm(10000),
  x = rnorm(10000))
sampleid <- sample(1:4, 10000, replace = TRUE)
```

算法实现

- 从机的函数
  - 每台从机的工作是相同的，对于分配给自己的数据，运行 lm 函数，然后返回 x 的估计值和 P 值。
  - 对于每台从机只需要根据自己的 ID 选择对应组号的数据进行运算即可。
  - 该函数并没有参数，但是函数中包含了 lmdata、sampleid 和 slaveID 这三个外部环境的变量。
- 示例代码

```
slavefun <- function() {
  tmp.data <- lmdata[which(sampleid == slaveID), ]
  tmp.model <- lm(y~x+0, data = tmp.data)
  tmp.sum <- summary(tmp.model)
 return(c(Est=tmp.sum$coefficients["x","Estimate"],
        P = tmp.sum$coefficients["x", "Pr(>|t|)"]))
}
```

执行程序

- 广播数据
  - mpi.bcast.Robj2slave 用广播的方式来传入数据和函数。
  - mpi.bcast.cmd 用广播的方式来传入命令。

```
mpi.bcast.Robj2slave(lmdata)
mpi.bcast.Robj2slave(sampleid)
mpi.bcast.Robj2slave(slavefun)
mpi.bcast.cmd(slaveID <- mpi.comm.rank())
```

- 执行程序

```
res <- mpi.remote.exec(slavefun(), simplify = FALSE)

## $slave1
## [1] 0.01214 0.54001
## $slave2
## [1] 0.01954 0.32362
## $slave3
## [1] 0.01802 0.37171
## $slave4
## [1] -0.02525  0.20676
```

目 录

snow 框架

- snow 包简介
  - 基于 R 语言的一个并行框架，可以支持多种通信方式和并行机制，包括基于 MPI 的集群。
  - R 内置的 `parallel` 将其整合后进行了简化，比如只支持 PSOCK 和 Fork 机制。
  - `snow` 本质上是个简化的并行框架，采用 Scatter/Gather 范式，Worker 之间不能通信。
  - 单独安装 `snow` 包可以应用一些更复杂的功能。
- Scatter/Gather
  - Scatter：Manager 把要做的计算分解成块，然后把块发送（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 把结果接受（收集）起来，整合结果来解决最开始的问题。

并行计算与并行框架

- MPI 与并行计算
  - MPI 可以适应各种硬件环境，不再局限于单机多核或者多 CPU 的机器，可以扩展到由很多台不同类型机器组成的集群，对硬件的要求比较低。
  - MPI 的各节点之间可以互相通信，因此在算法上可以非常灵活地处理各种问题。
  - 通过 MPI 可以精确地控制到每个节点、每个任务，充分实现并行的需求。
- 并行框架的价值
  - 对于一次性的工作，要考虑人的时间和机器时间之间的权衡，S 语言的一个基本理念是“人类的时间永远比机器的时间更宝贵”。
  - 对于需要多次运行的任务，如果效率非常重要，可以尽可能地灵活编程，将并行的效率发挥到极致。
  - 对于需要多次解决并行任务的工作，需要一套编程简单的框架，可以牺牲一定程度的效率。

目 录

目 录

问题描述

- 计算素数
  - 对于自然数 n，返回一个从 2 到 n 之间所有素数构成的向量。
  - 寻找素数的算法在密码学中很常用。
- 算法思路
  - 我们使用一个流水线的方法进行计算，也称为埃拉托色尼筛。
  - 对于一个 2 到 n 的数列，我们先划掉所有 2 的倍数，然后划掉所有 3 的倍数，然后是 5、7 等后续素数的倍数，最后没有划掉的数字就是素数。
  - 我们采用一个向量 `divisors` 作为“素数泵”，可以使用非并行的算法提前计算。可以证明如果一个数字有一个大于 $\sqrt{n}$ 的除数，那么一定也有一个比 $\sqrt{n}$ 更小的除数，因此可以把所有小于 $\sqrt{n}$ 的素数作为素数泵。
  - 在 `divisors` 的基础上可以进行并行计算。首先将数据分段，每一段数据可以交给一些节点依次筛选不同的素数，这样相邻的两个节点之间会进行信息的交互，适合用消息传递机制的并行方式来解决。

串行计算素数

- 代码说明
  - serprime 用来计算长度为 n 的自然数向量中包含的素数。
  - 程序的思路是遍历 $\sqrt{n}$ 以内的数，判断所有的数是否能被其整除，如果能整除的就剔除。由于大于 $\sqrt{n}$ 的能整除的数一定可以找到一个小于 $\sqrt{n}$ 的数相乘后能整除，所以在遍历 $\sqrt{n}$ 以内的数的时候就已经剔除了。
- 代码示例

```
serprime <- function(n) {
  nums <- 1:n
  prime <- rep(1,n)
  maxdiv <- ceiling(sqrt(n))
  for (d in 2:maxdiv) {
    if (prime[d]) {
      prime[prime!=0 & nums>d & nums%%d==0] <- 0
    }
  }
  nums[prime != 0 & nums >= 2]
}
```

判断是否能整除

- 代码说明
  - 函数 dosieve 用来判断 x 是否能够被 divs 整除。
  - x 是待剔除的向量。
  - divs 是一串数（通常为素数）构成的向量，程序对其进行遍历，判断其中的每一个数值是否能整除 x，将 x 中能被整除的数不断剔除。
  - 最终返回 x 中所有不能被 divs 整除的数构成的向量以及 divs 本身。
  - 该函数返回的结果在每个节点中作为消息传递到其他节点。
- 代码示例

```
dosieve <- function(x,divs) {
   for (d in divs) {
      x <- x[x %% d != 0 | x == d]
   }
   x
}
```

从机运行的函数

- 代码说明
  - dowork 在从机运行，每个节点通过 mpi.comm.rank 函数得到自己的序号。
  - 参数 n 代表要搜索的序列的个数，必须是偶数。divisors 是一个素数向量，是埃拉托色尼筛操作的基础，每个节点遍历一段素数，依次将筛选后的向量传给后面的节点。
  - 参数 msgsize 未使用，是为了和主机函数保持一致。
- 代码示例

```
dowork <- function(n,divisors,msgsize) {
  me <- mpi.comm.rank()
  lastnode <- mpi.comm.size()-1
  ld <- length(divisors)
  tmp <- floor(ld / lastnode)
  mystart <- (me-1) * tmp + 1
  myend <- mystart + tmp - 1
  if (me == lastnode) myend <- ld
  mydivs <- divisors[mystart:myend]
  if (me == lastnode) out <- NULL
```

从机运行的函数

```
  repeat {
    msg <- mpi.recv.Robj(tag=0,source=me-1)
    if (me < lastnode) {
      if (!is.na(msg[1])) {
        sieveout <- dosieve(msg,mydivs)
        mpi.send.Robj(sieveout,tag=0,dest=me+1)
      } else {
        mpi.send.Robj(NA,tag=0,dest=me+1)
        return()
      }
    } else {
      if (!is.na(msg[1])) {
        sieveout <- dosieve(msg,mydivs)
        out <- c(out,sieveout)
      } else {
        mpi.send.Robj(out,tag=0,dest=0)
        return()
      }
    }
  }
}
```

从机运行的函数

- 程序思路
  - 对于每一次执行，首先计算需要处理的素数序列的长度 tmp，然后根据每个节点的序号计算出该节点要处理的素数序列 mydivs。
  - 此后执行一个 repeat 循环，每次从前一个节点中接受函数 mpi.send.Robj 发送的信息 msg（一个向量）。
  - 如果当前节点不是最后的节点：
    - 如果接受的数据不为 NA，则调用 dosieve 函数来筛选 msg 中不能被 mydivs 整除的数，将剩余数据用 mpi.send.Robj 函数发送到下一个节点。
    - 如果接受的数据为 NA，说明没有后续数据了，将 NA 用 mpi.send.Robj 函数传递到下一个节点，并返回空值。
  - 如果当前节点是最后的节点：
    - 如果接受的数据不为 NA，则继续计算，并合并结果（每一步计算都会汇总到最后的节点）。
    - 如果接受的数据为 NA，说明没有后续数据了，将结果用 mpi.send.Robj 函数发送回主机。

主机运行的函数

- 代码说明
  - 函数 primepipe 在主机运行，参数 n 和 divisors 与 dowork 的一样，msgsize 代表块的大小。
- 代码示例

```
primepipe <- function(n,divisors,msgsize) {
   mpi.bcast.Robj2slave(dowork)
   mpi.bcast.Robj2slave(dosieve)
   mpi.bcast.cmd(dowork,n,divisors,msgsize)
   odds <- seq(from=3,to=n,by=2)
   nodd <- length(odds)
   startmsg <- seq(from=1,to=nodd,by=msgsize)
   for (s in startmsg) {
      rng <- s:min(s+msgsize-1,nodd)
      mpi.send.Robj(odds[rng],tag=0,dest=1)
   }
   mpi.send.Robj(NA,tag=0,dest=1)
   lastnode <- mpi.comm.size()-1
   c(2,mpi.recv.Robj(tag=0,source=lastnode))
}
```

主机运行的函数

- 注意事项
  - msgsize 代表每一次发送的信息（待搜索的向量）的大小，如果太小的话比较消耗通信的资源，太大的话失去了并行的意义。
  - 函数 mpi.bcast.cmd 实现了一种非阻塞调用，无需等待返回结果即可进行下一步。
- 程序思路
  - 首先使用 mpi.bcast.Robj2slave 函数将计算所需要的函数发布到每个节点。
  - 然后使用 mpi.bcast.cmd 函数启动每个从机，要求他们运行 dowork 函数。
  - 将所有奇数值根据 msgsize 的大小分成块，每块向量记为 rng，使用 mpi.send.Robj 将其全部发送到第 1 个节点。完成后发送 NA 作为结束标记。
  - 最终接收到最后一个节点的返回值之后，加上数字 2，一起返回。

执行运算

- 代码说明
  - 分别执行串行和并行函数，比较运行时间。
- 代码示例

```
system.time(serprime(1000000))

##    user  system elapsed
##    3.96    0.88    4.85

library(Rmpi)
system.time(primepipe(1000000, dvs, 100))

##    user  system elapsed
##  168.09    0.01  168.94

system.time(primepipe(1000000, dvs, 1000))

##    user  system elapsed
##   17.02    0.03   17.21

system.time(primepipe(1000000, dvs, 10000))

##    user  system elapsed
##    1.94    0.00    2.21
```

目 录

性能的改进

- 延迟和带宽
  - 从之前的示例中可以发现 msgsize 对性能的影响很大。
  - 如果 msgsize 太小则会有更多的数据块，因此在网络延迟上会花费更多时间。
  - 如果 msgsize 太大则会使并行程度下降，无法进行延迟隐藏（等待延迟时进行计算）。
- 负载均衡
  - 我们的例子中，每个节点越往后处理的数据量越少，因为供筛选的数越来越稀疏，后面的节点处理的工作量会少很多。
  - 可以使用代码让向量 divisors 分配成不同的大小，把较大的块分配给靠后的进程。
- 内存分配
  - 如果使用 mpi.send 和 mpi.recv 来代替 mpi.send.Robj 和 mpi.recv.Robj，会更加高效。比如 mpi.recv.Robj 函数，每次迭代时都会自动进行内存分配。而 mpi.recv 是更基础的接受操作，可以为其设置一个缓冲空间，让每次迭代都使用同一块内存，可以提高效率。

消息传递的性能细节

- 阻塞与非阻塞
  - 以函数 `mpi.send` 为例，将数据 `x` 发送到某个节点，在大部分的 MPI 实现中，当 `x` 的空间可以重用时（比如将 `x` 的内容复制到操作系统的空间后）返回函数值。但是有些 MPI 的实现是会一直等到目标节点接受后才返回。由于网络延迟，这两种 MPI 的实现之间可能会有很大的性能差别。
  - 通常 MPI 提供了非阻塞的发送和接收函数，比如 `mpi.isend` 和 `mpi.irecv`，可以让代码发起一个发送或接收操作后进行其他的工作，之后再回来通过 `mpi.test` 等函数来检查操作是否完成。
- 死锁问题
  - 如果在 MPI 的实现中，发送操作被阻塞直到对应的接收完成，那么这段代码可能会造成死锁问题，意味着两个进程被卡死，都在等待对方。
  - 在编程的过程中，要注意消息之间的顺序，防止死锁问题。
  - 也可以使用非阻塞操作，但这样会使得代码更复杂。

Thank you!

李舰 Email: jian.li@188.com