并行机器编程

Norm Matloff（著）
加州大学戴维斯分校

寇强（译）
华南统计科学研究中心
印第安纳大学
新浪微博：Gossip_useR

GPU、多核、集群

http://heather.cs.ucdavis.edu/~matloff/probstatbook.html
本书更新网址：

英文版：http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf
中文版：https://github.com/thirdwing/ParaBook

作者和译者尽了最大努力，但书中错误在所难免
如果对翻译有任何疑问，请随时通过邮件kouqiang@mail3.sysu.edu.cn联系我

关于本书

为什么本书和其它的并行编程书籍不同呢？原因在于我们主要关注在实现层面：

- 这里几乎没有理论内容，诸如 O() 分析、最大理论加速、PRAM、有向无环图 (DAG) 等等。
- 书中使用的都是真实代码。
- 我们使用的都是主流的并行平台，包括 OpenMP、CUDA 和 MPI，而没有使用其它仍处于实验阶段的语言。
- 关于性能的主题——通信延迟、内存/网络连接、负载均衡等，在全书中交叉进行，并且都是在特定平台或应用的层面进行讨论的。
- 相当关注调试技术。

书中使用的主要编程语言 C/C++，但也使用了一些 R 代码。R 已经是最主流的用于数据分析的语言。作为一门脚本语言，R 可以用于快速原型构建。在本书中，我用 R 将可以一些例子表述得远远比用 C/C++ 要简洁，从而使得学生更容易的理解所使用的并行计算原则。出于同样的原因，学生们也可以更容易地编写并行代码，更加集中精力于这些原则之上。另外，R 也有相当丰富的并行库。

我们假设学生在编程方面是有相当经验的，并有包括线性代数在内的数学背景。附录里回顾了本书所需要的数学知识。另一个附录提供了不同系统问题的概述，包括进程调度和虚拟内存等。

需要特别说明的是，书中多数代码没有进行优化。我们主要关注的是技术和语言使用的清晰明了。然而，有很多影响速度的因素值得讨论，比如高速缓存一致性问题、网络延迟、GPU 内存结构等等。

这里展示了你可以如何使用书中的代码：本书使用 LaTeX 排版，原始的**.tex** 文件可以在 `http://heather.cs.ucdavis.edu/~matloff/158/PLN`下载。请直接下载相关文件（文件名应该足够明确），之后使用一个文本编辑器进行裁剪，从而得到感兴趣的代码。

为了向学生展示研究和教学的相互促进关系，我会时不时引用我的一些研究工作。

如同我的其它开源图书，本书是在**不停变动**中的。我会继续添加新的主题、新的示例等等，当然也会修补漏洞和改善说明。由于这个原因，所以保存本书最新版本的链接，`http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`，比保存一份拷贝更好。

同样出于这个原因，我非常希望得到反馈。这里我希望感谢 Stuart Ambler、Matt Butner、Stuart Hansen、Bill Hsu、Sameer Khan、Mikel McDaniel、Richard Minner、Lars Seeman、Marc Sosnick 和 Johan Wikström 的评论。特别感谢 Hsu 教授为我提供了高级的 GPU 设备。

各位可能对我另一本关于概览和统计的开源图书感兴趣，可以在 `http://heather.cs.ucdavis.edu/probstatbook`下载。

作者简介

Norm Matloff 博士是加州大学戴维斯分校计算机教授，曾任同校的统计学教授。他曾是硅谷的一名数据库软件开发人员，也曾作为统计咨询师为 Kaiser Permanente Health Plan 等公司工作过。

Matloff 博士生于 Los Angeles，在 East Los Angeles 和 San Gabriel Valley 长大。他从 UCLA 得到了纯数学的博士学位，研究方向为概率论和统计他在计算机科学和统计学方面发表了大量论文，现在的研究方向是并行处理、统计学和回归方法。

Matloff 博士曾是 UNESCO 下的数据库安全国际委员会，IFIP Working Group 11.3 成员。他是戴维斯分校统计系的创始人之一，并参与了计算机系的建立。他在戴维斯分校被授予 Distinguished Teaching Award and Distinguished Public Service Award。

Matloff 博士是两本书的作者，并编写了大量广泛使用的网络教程，涉及 Linux 和 python 语言。他和 Peter Salzman 博士是《软件调试的艺术》[①]一书的作者。Matloff 博士关于 R 语言的《R 语言编程艺术》[②]一书已于 2011 年出版。他的新书，*Parallel Computation for Data Science* 会于 2014 年出版。他还写作了很多开源图书，包括 *From Algorithms to Z-Scores: Probabilistic and Statistical Modeling in Computer Science*（`http://heather.cs.ucdavis.edu/probstatbook`），和《并行机器编程》（`http://heather.cs.ucdavis.edu/~matloff/ParProcBook.pdf`）。

[①]译者注：The Art of Debugging with GDB, DDD, and Eclipse，中文版已由人民邮电出版社出版
[②]译者注：The Art of R Programming，中文版已由机械工业出版社出版

第 10 章　R 并行处理入门

10.1　为什么要在本书中用 R 语言?

在本书的其它章节里，C/C++ 依然是我们的主要语言，但我们也提供很多 R 语言的示例。为什么要用 R 呢?

- R 是最广泛使用的用于统计分析和数据处理的编程语言。在现今这个大数据时代，人民已经开发了相当数量的用于并行计算的 R 扩展包。特别地，**parallel** 扩展包现在已经是 R 基础包的一部分。
- R 语言的广泛使用，从 Google 设置了其内部的 R 语言规范一事就可见一斑[①]。现在 Oracle 也把 R 包含了自己的大数据分析方案中。
- 对于展示各种各样的并行算法，R 非常方便。这点的主要原因在于 R 内置了向量、矩阵和复数类型。

Python 也有很多并行库，比如 **multiprocessing**。关于 Python 的并行话题，我们会在第16章里讨论。

本章的示例会保持尽量简单。但 R 中的并行计算也可以应用到非常庞大而复杂的问题上。在附录B中，有一个 5 分钟的 R 快速入门。阅读时请牢记 R 中 list 结构。

R 中进行并行计算的关键就是——list 结构的操作。许多 R 的并行计算扩展包都非常依赖于 R 中的 list 结构。输入输出的参数和返回值经常都采用 list 的形式。读者可能有兴趣参考一下附录B中的相关内容。

10.2　R 和易并行问题（Embarrassing Parallel Problems）

需要注意的是，R 的并行扩展包一般只能处理易并行问题。正如在 2.3节中定义的，这些问题不仅容易并行化，而且信息传递的需求很少[②]。如我们所知，一般只有易并行问题会有很好的表现，但在 R 中情况尤其如此，原因如下。

R 语言的函数式编程的本质意味着，任何对一个向量或矩阵的元素的写入操作，比如

```
1 x[3] <- 8
```

都会重写整个向量或矩阵[③]。虽然有些例外（随着 R 版本更新，例外可能越来越多），但一般来说我们必须承认 R 中并行的向量和矩阵代码代价很高[④]。

对于不易并行的问题，大家应该考虑用 R 调用并行的 C 代码，这点会在10.9 节中讨论。

---

①个人角度来讲，我并不喜欢这些代码规范，我更喜欢我自己的。但从 Google 设置自己的 R 语言规范可以看出他们对 R 的重视程度。
②后面的要求把很多迭代算法排除在外了，尽管它们很容易并行化。
③R 中的元素赋值是一个函数调用，上面这个例子的参数分别为 `x`、3 和 8。
④R 中新的引用类（Reference class）可能会对此有所改变。

10.3 一些 R 的并行扩展包

这里我们列举了一些 R 的并行扩展包：

- Message-passing 或 scatter/gather (7.4节)：**Rmpi**、**snow**、**foreach**、**rmr**、**Rhipe**、**multicore**[5]、**rzmq**
- 内存共享：**Rdsm**、**bigmemory**
- GPU：**gputools**、**rgpu**

大家可以从 `http://cran.r-project.org/web/views/HighPerformanceComputing.html`找到更加详尽的列表。

从 2.14 版本开始，R 默认包括了由 **snow** 和 **multicore** 构成的 **parallel** 扩展包。（早期版本可能需要分别下载。）正是因为如此，二者都在范围之内。另外，我们也会讨论 **Rdsm/bigmemory** 和 **gputools**。

10.4 安装和载入这些扩展包

安装：

需要注意的是，如果你使用的是 2.14 版或更高版本的 R，你已经安装了 **snow** 和 **multicore**。一般来说，除了 **rgpu**，其它所有扩展包都可以从 R 官方的代码仓库 CRAN（`http://cran.r-project.org`）下载。这里以 **snow** 为例：

加入你想把它安装在**/a/b/c/** 目录下。最简单的方法就是使用 R 的函数：

```
> install.packages("snow","/a/b/c/")
```

这会将 **snow** 安装在**/a/b/c/snow** 目录下。

之后你需要将目录**/a/b/c**（不是**/a/b/c/snow**）加到你的 R 搜索路径中。我推荐大家在自己 home 目录下的**.Rprofile** 文件（这是 R 的启动设置文件）中添加这样一行。

```
.libPaths("/a/b/c/")
```

在一些情况下，由于所需库的位置原因，你可能需要手动安装一个 CRAN 上的扩展包。这一点请参考下面的10.8.1节和10.8.3节。

载入一个扩展包：

通过调用 **library()** 来载入一个扩展包。例如，载入 **parallel**，可以使用：

```
> library(parallel)
```

10.5 R 中的 snow 扩展包

snow 最大的优点在于其简单。其概念和实现都非常简单，能出错的地方不多。因此，它可能是现在使用最广泛的 R 并行包。

snow 扩展包可以直接通过 network socket 运行（由于用户只需要安装 **snow**，着可能是最常见的用法），也可以运行于 **Rmpi**（R 的 MPI 接口）、PVM 或 NWS 之上。

[5] **multicore** 扩展包运行于多核内存共享的平台之上，但在读写过程中并不共享数据。

它也可以在一个 scatter/gather 模型（7.4节）下进行操作。正如 R 中的 **apply()** 函数会将同样的函数作用于一个矩阵的每行上（见下面的示例），**snow** 中的 **parApply()** 会在多台机器上并行地完成类似的操作；不同的机器会操作不同的行。（除了使用多台机器，我们也可以在多核的机器上运行多个 **snow** client。）

10.5.1 使用

在使用

```
> library(snow)
```

载入 **snow** 之后，通过调用 **snow** 中的 **makeCluster()** 函数，我们可以设置一个 **snow** 集群。该函数的 **type** 参数用于选择网络平台，诸如 “MPI” 或 “SOCK”。后者用于将 **snow** 运行于其自己创建的 TCP/IP sockets 之上，而不是使用 MPI 。

在这个例子里，我在名为 **pc48** 和 **pc49** 的电脑上使用 “SOCK” 选择，以这种方式设置集群[6]：

```
> cls <- makeCluster(type="SOCK",c("pc48","pc49"))
```

需要注意的是上面的 R 代码在名为 **pc48** 和 **pc49** 的机器上设置了**工作节点**；这和**管理节点**相区别，管理节点运行于执行 R 代码的机器上。

如果你想把工作节点和管理节点同时运行在同一台机器上（特别是在一台多核的机器上），需要使用 **localhost** 作为机器名。

还有其它很多可选的参数。一个你可能觉得非常有用的是 **outfile**，它会把调用的结果记录在名为 **outfile** 的文件里。这在调用失败进行 debug 时非常有用。

10.5.2 示例：使用 parApply() 进行矩阵向量相乘

为了介绍 **snow**，让我们考虑一个简单的矩阵向量相乘的简单示例。我是指一个测试矩阵如下：

```
> a <- matrix(c(1:12),nrow=6)
> a
     [,1] [,2]
[1,] 1 7
[2,] 2 8
[3,] 3 9
[4,] 4 10
[5,] 5 11
[6,] 6 12
```

我们会将向量 $(1,1)^T$ （T 这里表示转置）和矩阵相乘。在这个简单的示例，我们当然可以直接完成：

```
> a %*% c(1,1)
     [,1]
[1,] 8
```

[6] 如果你使用的是一个文件共享系统的电脑集群，尽量保证 R 的安装路径一致，以避免问题。

```
[2,] 10
[3,] 12
[4,] 14
[5,] 16
[6,] 18
```

但是让我们看看如何使用 R 的 **apply()** 来完成它。尽管这仍是顺序执行，但这为我们扩展到并行计算提供了便利。

R 的 **apply()** 函数调用一个用户定义的标量函数作用于用户指定的矩阵的每一行（或每一列）。为了将 **apply()** 用于这里的矩阵向量相乘问题，我们定义一个点积的函数：

```
> dot <- function(x,y) {return(x%*%y)}
```

现在调用 **apply()**：

```
> apply(a,1,dot,c(1,1))
[1] 8 10 12 14 16 18
```

这个调用将函数 **dot()** 作用于矩阵 **a** 的每一行（这个可以从 1 看出，2 意味着每一列）；每一行都将作为 **dot()** 的第一个参数，而 c(1,1) 会作为第二个参数。换言之，**dot()** 的第一次调用就是

```
dot(c(1,7),c(1,1))
```

snow 中的 **parApply()** 函数将 **apply()** 扩展到并行计算。我们把它用于将我们的矩阵相乘问题并行化，运行在我们名为 **cls** 的集群之上：

```
> parApply(cls,a,1,dot,c(1,1))
[1] 8 10 12 14 16 18
```

parApply() 所作的就是将矩阵每一行发送给每一个节点，同时发送的还由函数 **dot()** 和参数 **c(1,1)**。每个节点将 **dot()** 作用到接收的行上，之后将结果返回给管理节点。

R 的 **apply()** 函数一般只用于变量值的情形，这意味着 **apply(m,i,f)** 调用中的函数 **f()** 的返回值是标量。如果 **f()** 的返回值是向量值，那返回的会是一个矩阵而不是一个向量，矩阵里的每一列是 **f()** 作用于 **m** 的一列或一行的结果。**parApply()** 也同样如此。

10.5.3 snow 中的其它函数：clusterApply()、clusterCall() 等

上一节，我们介绍了 **parApply()** 函数。它可以这样调用

- **parApply()**:

```
parApply(cls,m,DIM,f,...)}
```

这个调用会把矩阵 **m** 的每一行分配到 **cls** 的各个工作节点，之后函数 **f()** 会被作用到每一行，省略号在这里表示可选参数。参数 **DIM** 为 1 时表示行操作，2 表示列操作。

返回值是一个向量（也可能是个矩阵，如上所述）。

snow 最大的有点在于其简单，因此并没有很多复杂的函数，但当然不止 **parApply()** 一个。这里列举了一些：

- **clusterApply()**:

这个函数可能是 **snow** 中被使用最频繁的函数。

```
clusterApply(cls,individualargs,f,...)}
```

这会使 **f()** 在 **cls** 中的每个节点上运行。这里的 **individualargs** 是一个 R 列表（如果是个向量，会被转换成列表）。当 **f()** 在集群中的节点 i 上被调用时，其参数如下所述：第一个参数是 **individualargs** 的第 i 个元素，或者说是 **individualargs**[[i]]；如果在调用时，是用了省略号所代表的（可选）参数，它们会作为第二、第三或更多的参数传递给 **f()**。

如果 **individualargs** 的元素数量大于集群中的节点数，那么 **cls** 会被循环使用（可以把它作为一个向量对待），所以多数或全部节点会在不止一个 **individualargs** 元素上调用 **f()**。返回值是一个 R 列表，其中第 i 个元素是 **f()** 作用于 **individualargs** 中第 i 个元素的结果。所以说，**individualargs** 列表又需要拆分并行计算的工作构成。

- **clusterApplyLB():**

  这是 **clusterApply()** 的负载均衡模式，目的在于解决我们在第2章中提到的性能问题。

  为了解释 **clusterApply()** 的两者形式的区别，假设我们的集群由 10 个节点，而我们有 25 个需要执行的任务（或者说 **individualargs** 的长度是 25）。如果使用 **clusterApply()**，会发生下列这些：

  – 前 10 个任务会被分配给工作节点，每个节点一个任务。
  – 管理节点会等这 10 个任务完成，之后再分配另外 10 个。
  – 管理节点会等这 10 个任务完成，之后在分配剩下的 5 个。
  – 管理节点会等这 5 个任务完成，之后返回 25 个结果。

  而是用 **clusterApplyLB()** 时，会按照下面这种方式执行：

  – 前 10 个任务会被分配给工作节点，每个节点一个任务。
  – 当由节点任务结束时，管理节点会马上行动，将第 11 个任务分配给这个节点，即使其它节点的任务还没完成。
  – 管理节点会继续照此工作，一旦一个节点任务完成，就会分配新的任务，知道所有任务完成。
  – 管理节点最后会返回 25 个结果。

  用第2章和 OpenMP 一章中的4.3.3节的说法，**clusterApply()** 使用了一种**静态**的调度策略，而 **clusterApplyLB()** 使用了一种动态策略；其中 chunk size 为 1。

- **clusterCall():**

  函数 **clusterCall(cls,f,...)** 将函数 **f()** 和省略号所代表的参数（如果有的话），发送到每个工作节点。在每个节点上，**f()** 会使用这些参数求值。返回值是一个 R 列表，第 i 个元素师第 i 个节点的计算结果。（一眼看上去，似乎每个节点都会返回同样的结果，但 `f()` 会使用每个节点特定的参数，从而返回不同的结果。）

- **clusterExport():**

  函数 **clusterExport(cls,varlist)** 会将名字出现在字符向量 **varlist** 中的变量拷贝到 **cls** 中的各个节点。你可以使用这个函数来避免从管理节点到工作节点开销巨大的数据传输。使用这个函数，你可以只传输数据集一次；通过在相应的变量上使用 **clusterExport()**，之后在工作节点上将其作为全局变量使用。同样地，返回值仍是个 R 列表，第 i 个元素师集群中第 i 个节点的计算结果。

默认情况下，被传输到工作节点的变量在管理节点上必须是全局变量。

需要特别注意的是，一旦你传输了一个变量，比如 **x**，从管理节点到各个工作节点上，各个拷贝和工作节点上的变量就是独立的了（各个拷贝之间也是相互独立的）。如果其中一个拷贝改变了，在其他拷贝中不会反应这些变化。

- **clusterEvalQ()：**

  函数 **clusterEvalQ(cls,expression)** 会在 **cls** 的各个节点上运行 **expression**。

10.5.4 示例：并行求和

现在让我们再看一个示例，我们用 **snow** 来进行并行求和。先从一个很简单的版本开始，之后再考虑复杂的版本。

```
parsum <- function(cls,x) {
  # 在节点上分配 x 的索引（实际上没有传输任何东西）
  xparts <- clusterSplit(cls,x)
  # 现在传输到节点上，并进行求和
  tmp <- clusterApply(cls,xparts,sum)
  # 现在将各个单独的加和合并得到结果
  tot <- 0
  for (i in 1:length(tmp)) tot <- tot + tmp[[i]]
  return(tot)
}
```

现在我们在一个有两个共走节点的集群 **cls** 上进行测试：

```
> x
[1] 1 2 3 4 5 6 5 12 13
> parsum1(cls,x)
[1] 51
```

结果不错。现在我们来想一下，这是如何完成的？

最基本的想法就是讲我们的向量分块，之后分配给工作节点。每个工作节点会把所分配的小块求和，再把结果返回给管理节点。管理节点会把这些结果求和，返回我们想要的求和的最终结果。

为了将我们的向量 **x** 分块并发给各个节点，我先来看 **snow** 中的函数 **clusterSplit()**。这个函数的输入是一个 R 向量，之后将其分块，分块的数量和工作节点数相同。

例如，在上面的两个工作节点的集群上，我们得到：

```
> xparts <- clusterSplit(cls,x)
> xparts
[[1]]
[1] 1 2 3 4

[[2]]
[1] 5 6 5 12 13
```

非常肯定的是，我们的列表 **xparts** 有在其一个元素中有 **x** 的一块，而另一个元素中有 **x** 的另一块。之后这两块被传输到两个工作节点上：

```
> tmp <- clusterApply(cls,xparts,sum)
> tmp
[[1]]
[1] 10

[[2]]
[1] 41
```

同样像 **snow** 中的其他函数一样，**clusterApply()** 会以列表的形式返回结果。这里我们将结果赋值给了 **tmp**。其内容如下

```
> tmp
[[1]]
[1] 10

[[2]]
[1] 41
```

也就是 **x** 每一小块的和。

为了得到最后的结果，我们不能简单地在 **tmp** 上使用 R 中 **sum()** 函数：

```
> sum(tmp)
Error in sum(tmp) : invalid 'type' (list) of argument
```

这是因为 **sum()** 接受的是向量，而不是列表。所以我们自己写一个循环来把结果加起来：

```
tot <- 0
for (i in 1:length(tmp)) tot <- tot + tmp[[i]]
```

需要注意的一点是，我们使用 `[[]]`来获取列表中的元素。

我可以通过调用 R 中的 **Reduce()** 函数来取代上面的循环，从而对代码进行优化。**Reduce()** 很像4.3.5节和8.6.3节中的 reduction 操作。（注意，这里是个串行操作，不是并行。）一般以 **Reduce(f,y)** 这种形式使用，它对函数 **f()** 和列表 **y** 进行如下操作

```
z <- y[1]
for (i in 2:length(y)) z <- f(z,y[i])
```

使用 **Reduce()** 可以使代码更紧凑可读，一些情况下还会提高执行效率（我们这里只有很少的项目进行相加，暂时还不用考虑效率）。而且，**Reduce()** 会将 **tmp** 从一个列表转换为向量，这就解决了我们直接对 **tmp** 使用 **sum()** 时的问题。

下面是新的代码：

```
parsum <- function(cls,x) {
  xparts <- clusterSplit(cls,x)
  tmp <- clusterApply(cls,xparts,sum)
```

```
  Reduce(sum,tmp) # implicit return()
}
```

需要说明的是，在 R 中，如果没有显式的 **return()** 语句，那最后求得的值会被作为返回值，这里是 **Reduce()** 的计算结果。

Reduce() 是一个非常便利的函数，特别是在和 **snow** 一起使用时。这里有一个我们把多个矩阵进行合并的示例：

```
> Reduce(rbind,list(matrix(5:8,nrow=2),3:4,c(-1,1)))
     [,1] [,2]
[1,] 5 7
[2,] 6 8
[3,] 3 4
[4,] -1 1
```

rbind() 函数只有两个参数，在这里我们有三个。通过使用 **Reduce()** 可以解决这个问题。

10.5.5 示例：对角分块矩阵求逆

假设我们有一个对角分块矩阵，比如

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 8 & 1 \\ 0 & 0 & 1 & 5 \end{pmatrix}$$

我们想对其求逆。这是个易并行问题：假如我们有两个处理器，我们可以很简单地让其中之一对第一个 2×2 子矩阵求逆，让另一个对第二个 2×2 子矩阵求逆，之后我们将两个逆矩阵放回原来的位置。

通讯的开销在这里不是很大，一个 n×n 矩阵求逆的时间复杂度为 $O(n^3)$，而通讯只有 $O(n^2)$。

现在我们讨论一下用于分块对角矩阵求逆的 **snow** 代码。

```
# invert a block diagonal matrix m, whose sizes are given in szs;
# return value is the inverted matrix
bdiaginv <- function(cls,m,szs) {
  nb <- length(szs) # number of blocks
  dgs <- list() # will form args for clusterApply()
  rownums <- getrng(szs)
  for (i in 1:nb) {
     rng <- rownums[i,1]:rownums[i,2]
     dgs[[i]] <- m[rng,rng]
  }
  invs <- clusterApply(cls,dgs,solve)
  for (i in 1:nb) {
     rng <- rownums[i,1]:rownums[i,2]
     m[rng,rng] <- invs[[i]]
  }
```

```
  m
}

# find row number ranges for the blocks, returned in a # 2-column
# matrix; blkszs = block sizes
getrng <- function(blkszs) {
  col2 <- cumsum(blkszs) # cumulative sums function
  col1 <- col2 - (blkszs-1)
  cbind(col1,col2) # column bind
}
```

我们来检测一下：

```
> m
     [,1] [,2] [,3] [,4] [,5]
[1,] 1 2 0 0 0
[2,] 7 8 0 0 0
[3,] 0 0 1 2 3
[4,] 0 0 2 4 5
[5,] 0 0 1 1 1
> bdiaginv(cls,m,c(2,3))
          [,1] [,2] [,3] [,4] [,5]
[1,] -1.333333 0.3333333 0 0 0
[2,] 1.166667 -0.1666667 0 0 0
[3,] 0.000000 0.0000000 1 -1 2
[4,] 0.000000 0.0000000 -3 2 -1
[5,] 0.000000 0.0000000 2 -1 0
```

这里的 **szs** 参数，包含了分块的大小。由于我们只有一个 2×2 和一个 3×3 的块，分块的大小就是 2 和 3，因为在函数调用里使用 **c(2,3)**。

这里 **clusterApply()** 的使用和早先的例子很相似。代码中值得注意的地方是我们需要保存每一块在大矩阵中的位置。最后我们写了一个 **getrng()** 函数，用于返回不同块的起始和结束的行数。我们通过使用这个函数来设置 **clusterApply()** 的 **dg** 参数：

```
for (i in 1:nb) {
  rng <- rownums[i,1]:rownums[i,2]
  dgs[[i]] <- m[rng,rng]
```

大家要记得表达式 **m[rng,rng]** 会提取 **m** 的行和列出来，在这里就是第 i 块。

10.5.6 示例：Mutual Outlink

让我们考虑2.4.3节中的例子。我们有一个网络，比如 web 链接。对于其中的两个节点，比如两个网站，我可能对其 mutual outlink 感兴趣，也就是两个网站共同的对外链接。

下面的 **snow** 代码会计算整个网络中任意一对节点 mutual outlink 的均值。

```
# snow version of mutual links problem

library(snow)

mtl <- function(ichunks,m) {
   n <- ncol(m)
   matches <- 0
   for (i in ichunks) {
      if (i < n) {
         rowi <- m[i,]
         matches <- matches +
            sum(m[(i+1):n,] %*% as.vector(rowi))
      }
   }
   matches
}

# returns the mean number of mutual outlinks in m, computing on the
# cluster cls
mutlinks <- function(cls,m) {
   n <- nrow(m)
   nc <- length(cls)
   # determine which worker gets which chunk of i
   options(warn=-1)
   ichunks <- split(1:n,1:nc)
   options(warn=0)
   counts <- clusterApply(cls,ichunks,mtl,m)
   do.call(sum,counts) / (n*(n-1)/2)
}
```

对于 **m** 中的每一行，我们会计算其下面每一行中的 mutual link。为了在工作节点之间分配工作，我们可以如下使用 **clusterSplit()**

```
clusterSplit(cls,1:nrow(m))
```

但这会有一个在2.4.3节中讨论过的不均衡问题。比如我们有两个工作节点和 100 行。如果我们像上一节一样使用 **clusterSplit()**，第一个节点进行的比较工作会远比第二个节点多。

一个解决方案是在调用 **clusterSplit()** 之前，将行号随机打乱。另一方法，也是我们上面的代码中使用的，就是用 R 的 **split()** 函数。

那 **split()** 是做什么的？它根据第二个参数中设置的 “类别”，将第一个参数进行分块处理。我们来看这个示例：

```
> split(2:5,c('a','b'))
$a
[1] 2 4
```

```
4
5 $b
6 [1] 3 5
```

这里的种类是 a 和 b。**split()** 函数要求第二个参数和第一个参数长度相同，所以首先会对第二个参数进行 “循环” 处理成 a,b,a,b,a。之后会将 2 和 4 放入类别 a，将 3 和 5 放入类别 b。**split()** 函数最后会返回一个相应的列表。

现在我们再回到上面的 **snow** 示例，我们仍然假设在两个工作节点中分配 100×100 的矩阵 **m**，代码

```
1 nc <- length(cls)
2 ichunks <- split(1:n,1:nc)
```

会生成一个由两部分构成的列表，第一部分由奇数行构成，第二部分由偶数行构成。之后我们再使用

```
1 counts <- clusterApply(cls,ichunks,mtl,m)
```

就可以在两个工作节点间实现一个负载均衡了。

注意这个调用需要将 **m** 作为一个参数（作为函数 **mtl()** 的参数）。否则工作节点将没有可供使用的 **m**。另一个选择是使用 **clusterExport()** 来将 **m** 发送到工作节点，之后作为一个全局变量供 **mtl()** 使用。

另外，调用 **options()** 是为了让 R 在我们做 “循环” 时不发出警告。一般我们并不这么做，但这里为了使用 **split()** 的需要。

之后，为了得到输出的列表中各个元素的总和，我们可以再次使用 **Reduce()**，但由于 R 的多样性，我们也可以使用 **do.call()** 函数。这个函数的动能正如其函数名暗示的：它会把列表 **counts** 中的每个元素抽出，之后作为参数传递给 **sum()**！（一般来说，当我们需要调用一个特定的函数，但其参数的数目直到运行时才可以确定时，**do.call()** 是非常有用的。）

正如前面所说的，除了使用 **split()**，我们可以将行数随机打乱：

```
1 tmp <- clusterSplit(cls,order(runif(nrow(m))))
```

这会为每一行产生一个 (0,1) 之间的随机数，之后按此排序。比如说，如果第三个随机数是第 20 小的，第三个元素在 **order()** 的输出中会是 20。这可以找到矩阵 **m** 行号的一个随机排列。

10.5.7 示例：邻接矩阵变换

这是4.13节中代码的 **snow** 版本。回忆一下，问题如下：

假如我们有一个图的邻接矩阵

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{10.1}$$

其中行号和列号从 0 开始，而不是 1。我们想要将其转换为一个两列的矩阵用来展示连接数，如下

所示

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 0 \\ 3 & 1 \\ 3 & 2 \end{pmatrix} \tag{10.2}$$

比如说，在上面的邻接矩阵中，最右边的第二行有一个 1，这意味着在顶点 1 和顶点 3 直接存在一条边。这个在转换后的矩阵中以 (1,3) 表示。

下面是在 **snow** 中进行该计算的代码：

```
tg <- function(cls,m) {
  n <- nrow(m)
  rowschunks <- clusterSplit(cls,1:n) # make chunks of row numbers
  m1 <- cbind(1:n,m) # prepend col of row numbers to m
  # now make the chunks of rows themselves
  tmp <- lapply(rowschunks,function(rchunk) m1[rchunk,])
  # launch the computation
  tmp <- clusterApply(cls,tmp,tgonchunk)
  do.call(rbind,tmp) # combine into one large matrix
}

# a worker works on a chunk of rows
tgonchunk <- function(rows) {
  # note: matrix space allocation not efficient
  mat <- NULL
  nc <- ncol(rows)
  for (i in 1:nrow(rows)) {
     row <- rows[i,]
     rownum <- row[1]
     for (j in 2:nc) {
        if (row[j] == 1) {
           if (is.null(mat)) {
              mat <- matrix(c(rownum,j-1),ncol=2)
           } else
              mat <- rbind(mat,c(rownum,j-1))
        }
     }
  }
  return(mat)
}
```

这里有什么新东西么？首先，由于我们需要最后的输出按字典序排列，我们需要保存原有每行的索引。所以我们需要在 **m** 中多添加一列：

```
m1 <- cbind(1:n,m) # prepend col of row numbers to m
```

其次，注意 **lapply()** 函数的使用。正如 **apply()** 会把一个特定的函数作用到矩阵的每一行（或每一列）上，**lapply()** 会把一个特定函数作用到列表的每一个元素上。输出结果依然是个列表。

在我们这里的例子中，我们需要将 **m** 按行分块传递给 **clusterApply()**，但后者要求我们必须传递一个列表。我们可以通过一个 **for** 来完成，一个一个地将分块添加进列表，但使用 **lapply()** 可以更加紧凑。

在最后，管理节点会接收新矩阵的很多部分，这些必须被整合起来。使用 **rbind()** 函数是很自然的想法，但我们仍然需要客服各个部分是 R 列表的问题。尽管 **Reduce()** 也可以完成，但用 **do.call()** 会更趁手。

需要注意的是，尽管在上一段中说使用 **rbind()** 是很自然的，但效率很低。这是因为 **rbind()** 会重新分配一个新的矩阵空间，这是个很浪费时间的操作。先分配 50 行空间，之后在构建矩阵时进行填充会是更好的选择。无论什么时候，我们用完了一个矩阵，我们都可以构建一个新的矩阵，之后把所有矩阵作为一个列表返回。

10.5.8 示例：设置节点 ID 和集群规模提示

让我们回忆一下，在 OpenMP 中有两个函数，**omp_get_thread_num()** 和 **omp_get_num_threads()**，分别用来报告一个线程的 ID 和线程总数。在 MPI 中，对应的函数时 **MPI_Comm_rank()** 和 **MPI_Comm_size()**。在 **snow** 中如果能有这样的函数（或功能），将是非常好的事情。这里的代码就是用于这个目的：

```
# sets a list myinfo as a global variable in the worker nodes in the
# cluster cls, with myinfo$id being the ID number of the worker and
# myinfo$nwrkrs being the number of workers in the cluster; called from
# the manager node
setmyinfo <- function(cls) {
   setmyinfo <- function(i,n) {
      myinfo <<- list(id = i, nwrkrs = n)
   }
   ncls <- length(cls)
   clusterApply(cls,1:ncls,setmyinfo,ncls)
}
```

是的，R 允许在函数中定义函数。顺便请注意超级赋值符<<-的使用，这个用了在全局层面进行赋值操作。

调用这个函数后，任何在一个工作节点上运行的代码代码都可以决定其节点 ID，比如在下面这样的代码中

```
if (myinfo$id == 1) ...
```

或者，我们也可以从管理节点传输代码到工作节点执行：

```
> setmyinfo(cls)
[[1]]
[[1]]$id
[1] 1

[[1]]$nwrkrs
[1] 2


[[2]]
[[2]]$id
[1] 2

[[2]]$nwrkrs
[1] 2

> clusterEvalQ(cls,myinfo$id)
[[1]]
[1] 1

[[2]]
[1] 2
```

第一个例子，由于 **clusterApply()** 有返回值，都会被打印出来。第二个例子中，调用

```
clusterEvalQ(cls,myinfo$id)
```

会使每个工作节点对表达式 **myinfo$id**进行求值；之后 **clusterEvalQ()** 返回在每个节点上的执行结果。

10.5.9 关闭集群

退出 R 之前，不要忘记使用 **stopCluster(clustername)** 来关闭集群。

10.6 multicore 扩展包

正如名字所暗示的，**multicore** 扩展包就是用于使用多核设备的计算能力的。这可能有点奇怪：既然 **snow** 既可以用于一个（物理）集群，也可以用于一个多核设备，而 **multicore** 只能在后者上使用，那用 **multicore** 的优势哪儿？答案是性能上的提高，这个在后面会解释。

这个扩展包的主函数是 **mclapply()**，其语法和 **snow** 中的 **clusterApply()** 很类似，也很类似地把任务分配给各个工作节点。

这里所说的工作节点，指的是同一台机器上的不同处理器。比如说，在一个四核的机器上运行 **multicore**，调用 **mclapply()** 会（默认）在你的机器上调用 4 个 R，并行地进行你的运算工作。其中每个 R 调用都使用和调用前的 R 一样的变量设置。因此所有的变量最初（注意这个修饰语）

是共享的，而不需要程序员采取特别的措施来将变量从管理节点分配到工作节点，这和 **snow** 相当不同。

这一切都是由 **mclapply()** 调用你操作系统中的 **fork()** 函数来完成的。（所以这仅限于类 Unix 系统，比如 Linux 和 MacOS。）这个 fork 过程是由 R 自己完成的，每个工作节点一个新的拷贝。

因此分配到 R 拷贝的工作节点会共享所有在 fork 发生时存在的变量（也包括你调用 **mclapply()** 时的局部变量）。所以你的代码不需要将这些变量拷贝到工作节点，工作节点会自动获取它们。但需要注意的是，这些变量只是在最初的时候是共享的，对其中一个拷贝的修改不会再其它拷贝中有所反应（包括最初的那个）。

从管理节点到工作节点的初始值拷贝是基于 **copy-on-write** 的，这意味着直到一个节点尝试获取数据，这份数据才会被拷贝过去。这个粒度（granularity）是在虚拟内存页（virtual memory page）层面（**??**节）上的。同样，这由操作系统处理，不是 R。

由于这个物理拷贝最终还是会由操作系统完成，所以 **multicore** 相对 **snow** 并没有很多人所想的那么有优势。然而，这可能在处理延迟方面由一定优势（**??**节）。有些情况下，不需要所有节点同时获取变量，所以可能一个节点在拷贝变量，而其余的在进行计算。.

还需要注意一点，**snow** 中，一个集群被设置好，会在每个 **snow** 函数调用中重复使用，而 **multicore** 与此不同，每一个 R 进程都在一个 **multicore** 函数被调用时从头开始。

10.6.1 示例：使用 multicore 进行邻接矩阵转换

和10.5.7节中的示例一样，而且实际上下面的 **tgonchunk()** 函数就是我们前面 **snow** 代码的修改版。

```
mclapply(starts,tgonchunk,m1,chunksize,mc.cores=ncores)
```

这个调用会将 **tgonchunk()** 函数作用于 **starts** 向量的每一个元素上（首先会被转换成 R 列表），其中 **m1** 和 **chunksize** 作为 **mclapply()** 的参数使用。

```
# transgraph problem, R multicore version

# arguments:
# m: the input matrix
# ncores: desired number of cores to use
tgmc <- function(m,ncores) {
  n <- nrow(m)
  chunksize <- floor(n/ncores)
  starts <- seq(1,n,chunksize)
  m1 <- cbind(1:n,m) # prepend col of row numbers to m
  tmp <- mclapply(starts,tgonchunk,m1,chunksize,mc.cores=ncores)
  do.call(rbind,tmp)
}

# a worker works on a chunk of rows
tgonchunk <- function(start,m1,chunksize) {
  # note: matrix space allocation not efficient
```

```
   outmat <- NULL
   end <- start + chunksize - 1
   nrm <- nrow(m1)
   if (end > nrm) end <- nrm
   ncm <- ncol(m1)
   for (i in start:end) {
      rownum <- m1[i,1]
      for (j in 2:ncm) {
         if (m1[i,j] == 1) {
            if (is.null(outmat)) {
               outmat <- matrix(c(rownum,j-1),ncol=2)
            } else
               outmat <- rbind(outmat,c(rownum,j-1))
         }
      }
   }
   return(outmat)
}
```

10.7 Rdsm

无论你在一个 NOW 网络还是一个多核机器上，我的 **Rdsm** 扩展包都可以作为一个多线程来使用。这是我在 2002 年开发的一个类似的 Perl 扩展包，PerlDSM[⑦]的扩展。**Rdsm** 的主要优势在于：

- 使用了一个内存共享的编程模型，正如在**??**节中所述，在并行编程社区中，一般认为这优于信息传递模型。
- 可以充分使用 R 的调试工具。

Rdsm 给了 R 程序员一个内存共享的视角，但实际上这些对象并没有共享。对象被储存在一个服务器上，通过网络端口获取[⑧]，从而使 R 程序员即使在 NOW 网络上也可以有一个类似多线程的视角。这里没有管理/工作节点的结构，所有的 R 进场都执行相同的代码。

Rdsm 中的共享对象，可以是 **dsmv** 和 **dsmm** 类中的数值向量或矩阵，也可以是 **dsml** 类中的 R 列表。为了效率，向量和矩阵与服务器的通讯是二进制的形式进行的，而列表进行了序列化。还有一个内置变量 **myinfo** 用于获取每一个进程的 ID 和进程总数，这和 **Rmpi** 中的 **mpi.comm.rank()** 和 **mpi.comm.size()** 返回的信息一样。

Rdsm 同样可以使用上面提到的 **install.packages()** 进行安装。**Rdsm** 提供了内置文档，不过最好还是要通读 **examples** 文件夹下的 **MatMul.R** 代码。里面提供了大量注释，希望可以作为这个扩展包的一个入门。

⑦N. Matloff, PerlDSM: A Distributed Shared Memory System for Perl, *Proceedings of PDPTA 2002*, 2002, 63-68.

⑧**Rdsm** 也可以在 **bigmemory** 扩展包中使用，见10.7.3节。

10.7.1 示例：使用 Rdsm 进行对角分块矩阵求逆

现在让我们来看如何将10.5.5节中的对角分块矩阵求逆使用 **Rdsm** 处理。

```
# invert a block diagonal matrix m, whose sizes are given in szs; here m
# is either an Rdsm or bigmemory shared variable; no return
# value--inversion is done in-place; it is assumed that there is one
# thread for each block

bdiaginv <- function(bd,szs) {
   # get number of rows of bd
   nrdb <- if(class(bd) == "big.matrix") dim(bd)[1] else bd$size[1]
   rownums <- getrng(nrdb,szs)
   myid <- myinfo$myid
   rng <- rownums[myid,1]:rownums[myid,2]
   bd[rng,rng] <- solve(bd[rng,rng])
   barr() # barrier
}

# find row number ranges for the blocks, returned in a 2-column matrix;
# matsz = number of rows in matrix, blkszs = block sizes
getrng <- function(matsz, blkszs) {
   nb <- length(blkszs)
   rwnms <- matrix(nrow=nb,ncol=2)
   for (i in 1:nb) {
      # i-th block will be in rows (and cols) i1:i2
      i1 <- if (i==1) 1 else i2 + 1
      i2 <- if (i == nb) matsz else i1 + blkszs[i] - 1
      rwnms[i,] <- c(i1,i2)
   }
   rwnms
}
```

相较于 **snow** 中的 11 行代码，这里主要的并行工作由这 4 行完成：

```
myid <- myinfo$myid
rng <- rownums[myid,1]:rownums[myid,2]
bd[rng,rng] <- solve(bd[rng,rng])
barr() # barrier
```

这也展示了内存共享编程模型相对信息传递模型的优势。

10.7.2 示例：Web Probe

在一般的编程社区中，一类主要的应用，甚至是在一个串行平台上，就是并行化 I/O。由于每一个 I/O 操作可以消耗很长时间（以 CPU 标准），如果可能的话，进行并行化是十分必要的。**Rdsm** 在 R 中提供了这样的功能。

下面的示例在一个很大的网站列表中循环进行，测量每次获取一个网站所用的时间。数据被储存在一个共享变量 **accesstimes** 中；前 **n** 个最近的获取时间被记录下来。每个 **Rdsm** 进程每次在处理一个网站。

这里的一个不寻常的特点是其中一个进程会马上退出，回到 R 的交换性命令行中。这就允许用户来检测搜集的数据。记住，共享的变量仍然可以被该进程获取。因此，当其他进程继续向 **accesstimes** 添加数据（每次添加时也进行一次删除），用户可以向退出的进程下命令来随着搜集工作，分析数据，比如说柱状图。

注意这里 lock/unlock 操作的使用，**Rdsm** 中使用了同样的名称。

```
# if the variable accesstimes is length n, then the Rdsm vector
# accesstimes stores the n most recent probed access times, with element
# i being the i-th oldest

# arguments:
# sitefile: IPs, one Web site per line
# ww: window width, desired length of accesstimes
webprobe <- function(sitefile,ww) {
  # create shared variables
  cnewdsm("accesstimes","dsmv","double",rep(0,ww))
  cnewdsm("naccesstimes","dsmv","double",0)
  barr() # Rdsm barrier
  # last thread is intended simply to provide access to humans, who
  # can do analyses on the data, typing commands, so have it exit this
  # function and return to the R command prompt
  # built-in R list myinfo has components to give thread ID number and
  # overall number of threads
  if (myinfo$myid == myinfo$nclnt) {
    print("back to R now")
    return()
  } else { # the other processes continually probe the Web:
    sites <- scan(sitefile,what="") # read from URL file
    nsites <- length(sites)
    repeat {
      # choose random site to probe
      site <- sites[sample(1:nsites,1)]
      # now probe it, recording the access time
      acc <- system.time(system(paste("wget --spider -q",site)))[3]
      # add to accesstimes, in sliding-window fashion
```

```
30          lock("acclock")
31          if (naccesstimes[1] < ww) {
32             naccesstimes[1] <- naccesstimes[1] + 1
33             accesstimes[naccesstimes[1]] <- acc
34          } else {
35             # out with the oldest, in with the newest
36             newvec <- c(accesstimes[-1],acc)
37             accesstimes[] <- newvec
38          }
39          unlock("acclock")
40       }
41    }
42 }
```

10.7.3 bigmemory 扩展包

Jay Emerson 和 Mike Kane 在我开发 **Rdsm** 的同时，开发了 **bigmemory** 扩展包；而我们互相都不知道这一点。

bigmemory 扩展包的目的不在于提供一个多线程环境。它的目的在于处理一个 R 的硬性限制：任何 R 对象都不能大于 $2^{31}-1$ 字节。即使你用一个有很大内存的 64 位机器，这个限制也是存在的。**bigmemory** 扩展包通过使用操作系统的调用在进程间设置共享内存，从而在多核机器上解决了这个问题[⑨]。

理论上讲，**bigmemory** 也可以用在多线程上，但其为包含这方面的机制。然而，**Rdsm** 可以和 **bigmemory** 一起使用，由于后者的高效，这也带来了优势。

在 **Rdsm** 中使用 **bigmemory** 变量非常简单：使用 **newbm()** 而不是 **cnewdsm()** 来创建共享变量即可。

10.8 R 和 GPU

未来几年中，（为了特定问题）将 GPU 的高效和 R 进行结合一定会让越来越多的程序员感兴趣。

现在，进行 GPU 开发的主流框架就是在 NVIDIA 的显卡上使用 CUDA。CUDA 是 C 的一个扩展。

如果你需要写自己的 CUDA 代码，那你可能要使用10.9节中的方法。但在多数情况下，你可以从 R 中 GPU 计算的两个主要扩展包，**gputools** 和 **rgpu**，找到你需要的功能。两个扩展包都主要处理线性代数操作。这小节剩余部分会讨论这两个扩展包。

10.8.1 安装

由于两个扩展包中链接到 CUDA 库的问题，你可能不能通过 **install.packages()** 来进行安装。我推荐的安装方法如下：

- 下载**.tar.gz** 格式的扩展包。

[⑨]通过使用操作系统将内存映射到文件上，这个扩展包也可以在分布式系统上使用

- 将扩展包解压缩，我们把产生的文件夹叫做 **x**。
- 假设你想把它安装到**/a/b/c**。
- 对 **x** 中的文件进行修改。
- 之后运行 `R CMD INSTALL -l /a/b/c x`。

更多细节会在后面的章节中讨论。

10.8.2 gputools 扩展包

为了安装 **gputools**，我从 CRAN 下载了源代码，并像前面提到的解压缩。我去掉了 **src** 文件夹下 **Makefile.in** 文件中的几个选项 `-gencode arch=compute_20,code=sm_20`。我还确定了 shell 的启动文件中包含了 CUDA 的执行路径和库路径，**/usr/local/cuda/bin** 和**/usr/local/cuda/lib**。

之后我运行了 **R CMD INSTALL**。我使用了 **gpuLm.fit()** 进行测试，R 中 **lm.fit()** 在 **gputools** 中的对应版本。

这个扩展包提供了多种线性代数操作，比如矩阵相乘、求解 $Ax = b$（矩阵求逆）和奇异值分解，以及一些需要大量计算的操作，比如线性/广义线性模型估计和层级聚类。

这里是如何求矩阵 **m** 平方的示例：

```
> m2 <- gpuMatMult(m,m)
```

gpuSolve() 函数和 R 中的 **solve()** 一样。调用 `gpuSolve(a,b)`会求解线性系统 $ax = b$，其中 **a** 是一个方阵，而 **b** 是一个向量。如果第二个参数缺失，会返回 a^{-1}。

10.8.3 rgpu 扩展包

为了安装 **rgpu**，我从 `https://gforge.nbic.nl/frs/?group_id=38`下载源代码并解压缩。之后我修改了 **Makefile** 文件中的几行 [10]

```
LIBS = -L/usr/lib/nvidia -lcuda -lcudart -lcublas
CUDA_INC_PATH = /home/matloff/NVIDIA_GPU_Computing_SDK/C/common/inc
R_INC_PATH = /usr/include/R
```

第一行是为了让系统找到 **-lcuda**，这点和 **gputools** 一样。第二行是为了 NVIDIA SDK 中的 **cutil.h** 文件，上面的是我的安装路径。

第三行中，我生成了一个 **z.c** 文件，其中只包含

```
#include <R.h>
```

一行，之后运行

```
R CMD SHLIB z.c
```

来看 R 的引用文件究竟在哪里。

在 2010 年 5 月是，**rgpu** 中的函数远少于 **gputools**。然而，**rgpu** 中一个很好的特性在于在进行矩阵运算时，不需要将中间结果从 device 内存返回到 host 内存，这是个开销很大的操作。这里展示了如何计算矩阵 **m** 的乘法，并加上自身：

[10] 译者注：请根据自己机器上的相应路径进行修改

```
> m2m <- evalgpu(m %*% m + m)
```

10.9 通过在 R 中调用 C 进行并行

并行 R 的目的在于比普通的 R 要快。但即使这个目的达到了，这也还是 R，也就是说，还是可能很慢。

人们总是必须决定花费多少精力在优化上面的。为了最快的速度，我们甚至都不应该用 C，而应该用汇编语言。类似地，也必须决定是纯粹用 R，还是用更快的 C。如果并行 R 给了你所需要的速度，那再好不过；如果速度不够，你应该考虑在主体仍用 R 的情况下，用 C 完成一部分工作。你会发现，在保持用 R 的方便的前提下，在用 C 处理代码的并行部分已经足够好了。

10.9.1 在 R 中调用 C

在 C 中，二维数组以行序存储，和 R 中列序相反。例如，如果我们有一个 3×4 的数组，第二行第二列中的元素在线性视角下是第 5 个元素，因为第一列中有 3 个元素，而这是第二列中的第二个元素。当然，还需要注意 C 的计数从 0 开始，而 R 从 1 开始。在写供 R 使用的 C 代码时，你必须考虑这些问题。

所有从 R 传递到 C 的参数都被作为 C 的指针。注意 C 函数自身必须返回 `void`。在 R/C 中传递的数值比如作为函数的参数，比如我们下面示例中的 `result`。

10.9.2 示例：矩阵的次对角线

作为一个示例，这里是用于求一个方阵次对角线的 C 代码[1]。代码被保存在一个名为 **sd.c** 的文件中：

```
// arguments:
// m: a square matrix
// n: number of rows/columns of m
// k: the subdiagonal index--0 for main diagonal, 1 for first
// subdiagonal, 2 for the second, etc.
// result: space for the requested subdiagonal, returned here

void subdiag(double *m, int *n, int *k, double *result)
{
  int nval = *n, kval = *k;
  int stride = nval + 1;
  for (int i = 0, j = kval; i < nval-kval; ++i, j+= stride)
     result[i] = m[j];
}
```

为了方便，你可以在一个命令行中运行 R 来编译它，这会调用 GCC：

```
% R CMD SHLIB sd.c
gcc -std=gnu99 -I/usr/share/R/include -fpic -g -O2 -c sd.c -o sd.o
```

[1] 感谢我的研究生助理 Min-Yu Huang，他完成了这个函数的早期版本。

```
gcc -std=gnu99 -shared -o sd.so sd.o -L/usr/lib/R/lib -lR
```

注意 R 向我们展示了它在调用 GCC 时所作的具体操作。这允许我们来做一些改动。

需要注意这只会生成一个动态链接库 **sd.o**，而不是一个可执行程序。（在 Windows 下会是**.dll**）所以，怎么来执行它？答案是使用 R 的 **dyn.load()** 函数来载入它。这里是一个示例：

```
> dyn.load("sd.so")
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
> k <- 2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
result=double(dim(m)[1]-k))
[[1]]
 [1] 1 6 11 16 21 2 7 12 17 22 3 8 13 18 23 4 9 14 19 24 5 10 15 20 25

[[2]]
[1] 5

[[3]]
[1] 2

$result
[1] 11 17 23
```

注意我们需要在调用时为 **result** 分配内存空间。从上面的结果来看，我们的函数在对应空间中放置的值是正确的。T

10.9.3 在 R 中调用 OpenMPI C 代码

由于 OpenMP 可以由 C 使用，这就使得其可以从 R 中调用。（关于 OpenMP 的详细讨论请见第4章。）

在10.9节中类似，代码被编译并载入到 R 会话，尽管有一些额外的步骤用于在调用 GCC 时设置 `-fopenmp` 参数（你需要手动运行，而不是使用 **R CMD SHLIB**）。

10.9.4 在 R 中调用 CUDA 代码

这里也适用同样的原则，但需要小心调用的库和类似的问题。

和上面一样，我们需要编译以动态链接库而不是可执行文件。这是下面要用的文件 **mutlinksforr.cu**，和编译用的命令：

```
pc41:~% nvcc -g -G -I/usr/local/cuda/include -Xcompiler
   "-I/usr/include/R -fpic" -c mutlinksforr.cu -o mutlinks.o -arch=sm_11
pc41:~% nvcc -shared -Xlinker "-L/usr/lib/R/lib -lR"
   -L/usr/local/cuda/lib mutlinks.o -o meanlinks.so
```

这会生成 **meanlinks.so**。之后我在 R 中进行测试：

```
> dyn.load("meanlinks.so")
```

```
> m <- rbind(c(0,1,1,1),c(1,0,0,1),c(1,0,0,1),c(1,1,1,0))
> ma <- rbind(c(0,1,0),c(1,0,0),c(1,0,0))
> .C("meanout",as.integer(m),as.integer(4),mo=double(1))
[[1]]
 [1] 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1 0

[[2]]
[1] 4

$mo
[1] 1.333333

> .C("meanout",as.integer(ma),as.integer(3),mo=double(1))
[[1]]
[1] 0 1 1 1 0 0 0 0 0

[[2]]
[1] 3

$mo
[1] 0.3333333
```

10.9.5 示例：Mutual Outlink

我们再次使用 2.4.3节中的 Mutual Outlink 示例。这里是 R/CUDA 版本的代码：

```
// CUDA example: finds mean number of mutual outlinks, among all pairs
// of Web sites in our set

#include <cuda.h>
#include <stdio.h>

// the following is needed to avoid variable name mangling
extern "C" void meanout(int *hm, int *nrc, double *meanmut);

// for a given thread number tn, calculates pair, the (i,j) to be
// processed by that thread; for nxn matrix
__device__ void findpair(int tn, int n, int *pair)
{ int sum=0,oldsum=0,i;
   for(i=0; ;i++) {
      sum += n - i - 1;
      if (tn <= sum-1) {
         pair[0] = i;
```

```
        pair[1] = tn - oldsum + i + 1;
        return;
      }
      oldsum = sum;
   }
}

// proc1pair() processes one pair of Web sites, i.e. one pair of rows in
// the nxn adjacency matrix m; the number of mutual outlinks is added to
// tot
__global__ void proc1pair(int *m, int *tot, int n)
{
   // find (i,j) pair to assess for mutuality
   int pair[2];
   findpair(threadIdx.x,n,pair);
   int sum=0;
   // make sure to account for R being column-major order; R's i-th row
   // is our i-th column here
   int startrowa = pair[0],
       startrowb = pair[1];
   for (int k = 0; k < n; k++)
      sum += m[startrowa + n*k] * m[startrowb + n*k];
   atomicAdd(tot,sum);
}

// meanout() is called from R
// hm points to the link matrix, nrc to the matrix size, meanmut to the output
void meanout(int *hm, int *nrc, double *meanmut)
{
    int n = *nrc,msize=n*n*sizeof(int);
    int *dm, // device matrix
        htot, // host grand total
        *dtot; // device grand total
    cudaMalloc((void **)&dm,msize);
    cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
    htot = 0;
    cudaMalloc((void **)&dtot,sizeof(int));
    cudaMemcpy(dtot,&htot,sizeof(int),cudaMemcpyHostToDevice);
    dim3 dimGrid(1,1);
    int npairs = n*(n-1)/2;
    dim3 dimBlock(npairs,1,1);
    proc1pair<<<dimGrid,dimBlock>>>(dm,dtot,n);
    cudaThreadSynchronize();
```

```
61     cudaMemcpy(&htot,dtot,sizeof(int),cudaMemcpyDeviceToHost);
62     *meanmut = htot/double(npairs);
63     cudaFree(dm);
64     cudaFree(dtot);
65 }
```

这份代码几乎没有进行优化。比如，我们应该在每一个 block 中使用不止一个线程。

10.10 调试 R 程序

R 内置的调试机制是首选，在还存在着其它选择。

10.10.1 文本编辑器

然而，如果你是一个 Vim 编辑器的粉丝，我开发了一个可以极大扩展 R 调试器的工具。请从 R 的 CRAN 上下载 **edtdbg**。Emacs 中也有类似的工具。

Vitalie Spinu 的 `ess-tracebug` 运行于 Emacs。它大体基于 `edtdbg`，但提供了更多的针对 Emacs 的特性。

10.10.2 IDE

我个人不是提倡使用 IDE，但的确有一些很优秀的 IDE。

REvolution Analytics，一家提供 R 咨询和再开发版本 R 的公司，他们提供了一个包含了很好的调试机制的 IDE。但它只可以在 Windows 上运行，而且必须安装 Microsoft Visual Studio。

StatET，一个基于 Eclipse 的跨平台 IDE 的开发者在 2011 年五月添加了调试工具。

RStudio，另一个跨平台的 IDE 的开发者，从 2011 年夏天也开始计划添加调试器[12]。

10.10.3 缺少命令行终端的问题

诸如 **Rmpi**、**snow**、**foreach** 和其它的并行 R 扩展包并未给每一个进程设置命令行终端，从而使得在工作节点上进行调试变得不可能。那我们怎样调试使用这些扩展包的程序呢？这里拿 **snow** 做个例子。

首先，需要调试调试每个工作节点上的函数，比如 10.5.6节 mutual outlink 示例中的 **mtl()** 函数。这时需要人为设置一下参数的值，之后使用 R 常规调试机制。

这可能有效。但 bug 很可能就出现在参数本身上，或者出现在我们设置它们的方式上。事情就变得困难了。由于 **print()** 在工作进程中无法工作，即使打印出诸如变量值的追踪信息都很难。**message()** 函数可能对一些扩展包有效；但如果无效，你需要自己使用 **cat()** 来将变量写到文件中。

Rdsm 支持全面的调试，它在每个进程中都有一个单独的命令行终端。

10.10.4 调试 R 所调用的 C 代码

如果像10.9节中，并行是通过在 R 中调用 C，生成一个动态链接库实现的，调试会更复杂一些。首先，需要在 GDB 下启动 R，之后载入需要调试的库。这是 R 的解释器会循环读取你发出的

[12]译者注：RStudio 中的调试功能已添加

R 命令。通过使用 ctrl-c 来跳出循环，这会让你返回 **GDB** 的解释器。之后在需要调试的 C 函数，比如上面例子中的 **subdiag()**，设置断点。最终，告诉 GDB 继续，它就在你的函数中暂停下来。这里展示了你的会话内容：

```
$ R -d gdb
GNU gdb 6.8-debian
...
(gdb) run
Starting program: /usr/lib/R/bin/exec/R
...
> dyn.load("sd.so")
```

10.11 本书中的其它 R 语言示例

见下列章节中的示例（一些是非并行的）：
12.5.4节、14.2.1节（非并行）和14.5.1节（非并行）。

- 线性等式的并行 Jacobi 迭代，12.5.4节。
- 1 维 FFT 的矩阵运算，14.2.1节（可以通过并行的矩阵相乘来并行化）。
- 2 维 FFT 的并行计算，14.4.1节。
- 图像平滑，14.5.1节。