并行机器编程

Norm Matloff（著）
加州大学戴维斯分校

GPU、多核、集群

本书更新网址：

http://heather.cs.ucdavis.edu/~matloff/probstatbook.html

英文版：http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf

中文版：https://github.com/thirdwing/ParaBook

作者尽了最大努力，但书中错误在所难免

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 debug 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），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]The Art of Debugging with GDB, DDD, and Eclipse，中文版已由人民邮电出版社出版

[②]The Art of R Programming，中文版已由机械工业出版社出版

目录

第 1 章　R 并行处理入门

1.1　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**??**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**??**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**??**中的相关内容。

1.2　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 **??**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。

[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。

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

```
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1.9 节中讨论。

1.3 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 (**??**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**。

1.4 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。

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。
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。

```
.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1.8.1节和1.8.3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)
```

1.5 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。

它也可以在一个 scatter/gather 模型（**??**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。）

1.5.1 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。

[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。

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。

1.5.2 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
[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。

1.5.3 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**??**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。

  By contrast, with **clusterApplyLB()**, the event flow will go this way:

  – The first 10 tasks will be sent to the workers, one task per worker.
  – When some node finishes, the manager will take action right away, sending the 11^{th} task to this node, even though the others aren't done.
  – The manager will continue in this fashion, giving each node a new task as soon as the node finishes its old one, until all the tasks are done.
  – The manager will then return the 25 results to the caller.

  In the language of Chapter **??**, and of Section **??** of our OpenMP chapter, **clusterApply()** employs a **static** scheduling policy, while **clusterApplyLB()** uses a dynamic one; chunk size is 1.

- **clusterCall():**

  The function **clusterCall(cls,f,...)** sends the function **f()**, and the set of arguments (if any) represented by the ellipsis above to each worker node, where **f()** is evaluated on the arguments. The return value is an R list, with the i^{th} element is the result of the computation at node i in the cluster. (It might seem at first that each node will

return the same value, but typically the `f()` will make use of variables special to the node, thus yielding different results.)

- **clusterExport():**
  The function **clusterExport(cls,varlist)** copies the variables whose names appear in the character vector **varlist** to each worker in the cluster **cls**. You can use this, for instance, to avoid constant shipping of large data sets from the master to the workers, at great communications costs. With this function, you are able to ship a quantity just once; you call **clusterExport()** on the corresponding variables, and then access those variables at worker nodes as (node-specific) globals. Again, the return value is an R list, with the i^{th} element is the result of the computation at node i in the cluster. By default, the variables to be exported must be global on the manager node.
  Note carefully that once you export a variable, say **x**, from the manager to the workers, their copies become independent of the one at the manager (and independent of each other). If one copy changes, that change will not be reflected in the other copies.
- **clusterEvalQ():**
  The function **clusterEvalQ(cls,expression)** runs **expression** at each worker node in **cls**.

1.5.4 示例：并行求和

Let's go over one more toy problem, in which we have **snow** do parallel summation. We'll do a simpler version, then a more advanced one.

```
parsum <- function(cls,x) {
   # partition the indices of x among the cluster nodes (nothing
   # is actually sent to them yet)
   xparts <- clusterSplit(cls,x)
   # now send to the nodes and have them sum
   tmp <- clusterApply(cls,xparts,sum)
   # now finish, combing the individual sums into the grand total
   tot <- 0
   for (i in 1:length(tmp)) tot <- tot + tmp[[i]]
   return(tot)
}
```

Let's test it on a two-worker cluster **cls**:

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

Good. Now, how does it work?

The basic idea is to break our vector into chunks, then distribute the chunks to the worker nodes. Each of the latter will sum its chunk, and send the sum back to the manager node. The latter will sum the sums, giving us the grand total as desired.

In order to break our vector **x** into chunks to send to the workers, we'll first turn to the **snow** function **clusterSplit()**. That function inputs an R vector and breaks it into as many chunks as we have worker nodes, just what we want.

For example, with **x** as above on a two-worker cluster, we get

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

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

Sure enough, our R list **xparts** has one chunk of **x** in one of its components, and the other chunk of **x** in the other component. These two chunks are now sent to our two worker nodes:

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

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

Again, **clusterApply()**, like most **snow** functions, returns its results in an R list, which we've assigned to **tmp**. The contents of the latter are

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

```
[1] 10

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

i.e. the sum of each chunk of **x**.

To get the grand total, we can't merely call R's **sum()** function on **tmp**:

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

This is because **sum()** works on vectors, not lists. So, we just wrote a loop to add everything together:

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

Note that we need double brackets to access list elements.

We can improve the code a little by replacing the above loop code by a call to R's **Reduce()** function, which works like the reduction operators we saw in Sections **??** and **??**. (Note, though, that this is a serial operation here, not parallel.) It takes the form **Reduce(f,y)** for a function **f()** and a list **y**, and essentially does

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

Using **Reduce()** makes for more compact, readable code, and in some cases may speed up execution (not an issue here, since we'll have only a few items to sum). Moreover, **Reduce()** changes **tmp** from an R list to a vector for us, solving the problem we had above when we tried to apply **sum()** to **tmp** directly.

Here's the new code:

```
parsum <- function(cls,x) {
   xparts <- clusterSplit(cls,x)
   tmp <- clusterApply(cls,xparts,sum)
   Reduce(sum,tmp) # implicit return()
}
```

Note that in R, absent an explcit **return()** call, the last value computed is returned, in this case the value produced by **Reduce()**.

Reduce() is a very handy function in R in general, and with **snow** in particular. Here's an example in which we combine several matrices into one:

```
> 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
```

The **rbind()** functions has two operands, but in the situation above we have three. Calling **Reduce()** solves that problem.

1.5.5 Example: Inversion of Block-Diagonal Matrices

Suppose we have a block-diagonal matrix, such as

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

and we wish to find its inverse. This is an embarrassingly parallel problem: If we have two processes, we simply have one process invert that first 2x2 submatrix, have the second process invert the second 2x2 submatrix, and we then place the inverses back in the same diagonal positions.

Communication costs might not be too bad here, since inversion of an nxn matrix takes $O(n^3)$ time while communication is only $O(n^2)$.

Here we'll discuss **snow** code for inverting block-diagonal matrices.

```
# 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
}
```

Let's test it:

```
> 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
```

Note the **szs** argument here, which contains the sizes of the blocks. Since we had one 2x2 block and a 3x3 one, the sizes were 2 and 3, hence the **c(2,3)** argument in our call.

The use of **clusterApply()** here is similar to our earlier one. The main point in the code is to keep track of the positions of the blocks within the big matrix. To that end, we

wrote **getrng()**, which returns the starting and ending row numbers for the various blocks. we use that to set up the argument **dg** to be fed into **clusterApply()**:

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

Keep in mind that the expression **m[rng,rng]** extracts a subset of the rows and columns of **m**, in this case the i^{th} block.

1.5.6 Example: Mutual Outlinks

Consider the example of Section **??**. We have a network graph of some kind, such as Web links. For any two vertices, say any two Web sites, we might be interested in mutual outlinks, i.e. outbound links that are common to two Web sites.

The **snow** code below finds the mean number of mutual outlinks, among all pairs of sites in a set of Web sites.

```
# 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) {
```

```
21    n <- nrow(m)
22    nc <- length(cls)
23    # determine which worker gets which chunk of i
24    options(warn=-1)
25    ichunks <- split(1:n,1:nc)
26    options(warn=0)
27    counts <- clusterApply(cls,ichunks,mtl,m)
28    do.call(sum,counts) / (n*(n-1)/2)
29 }
```

For each row in **m**, we will count mutual links in all rows below that one. To distribute the work among the worker nodes, we could have a call to **clusterSplit()** along the lines of

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

But this would presents a load imbalance problem, discussed in Section **??**. For instance, suppose again we have two worker nodes, and there are 100 rows. If we were to use **clusterSplit()** as in the last section, the first worker would be doing a lot more row comparisons than would the second worker.

One solution to this problem would be to randomize the row numbers before calling **clusterSplit()**. Another approach, taken in our full code above, is to use R's **split()** function.

What does **split()** do? It forms chunks of its first argument, according to "categories" specified in the second. Look at this example:

```
1 > split(2:5,c('a','b'))
2 $a
3 [1] 2 4
4
5 $b
6 [1] 3 5
```

Here the categories are 'a' and 'b'. The **split()** function requires the second argument to be the same length as the first, so it will first **recycle** the second argument to 'a','b','a','b','a'. The split will take 2,3,4,5 and treat 2 and 4 as being in categort 'a', and 3 and 5 to be category 'b'. The function returns a list accordingly.

Now coming back to our above **snow** example, and again assuming two workers and **m** 100x100, the code

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

produces a list of two components, with the odd-numbered rows in one component and the evens in the other. Our call,

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

then results in good load balance between the two workers.

Note that the call needed includ **m** as an argument (which becomes an argument to **mtl()**). Otherwise the workers would have no **m** to work it. One alternative would have been to use **clusterExport()** to ship **m** to the workers, at which it then would be a global variable accessible by **mtl()**.

By the way, the calls to **options()** tell R not to warn us that it did recycling. It doesn't usually do so, but it will for **split()**.

Then to get the grand total from the output list of individual sums, we could have used **Reduce()** again, but for variety utilized R's **do.call()** function. That function does exactly what its name implies: It will extract the elements of the list **counts**, and then plug them as arguments into **sum()**! (In general, **do.call()** is useful when we wish to call a certain function on a set of arguments whose number won't be known until run time.)

As noted, instead of **split()**, we could have randomized the rows:

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

This generates a random number in (0,1) for each row, then finds the order of these numbers. If for instance the third number is the 20^{th}-smallest, element 3 of the output of **order()** will be 20. This amounts to finding a random permutation of the row numbers of **m**.

1.5.7 Example: Transforming an Adjacency Matrix

Here is a **snow** version of the code in Section **??**. To review, here is the problem:

Say we have a graph with adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{1.1}$$

with row and column numbering starting at 0, not 1. We'd like to transform this to a two-column matrix that displays the links, in this case

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 0 \\ 3 & 1 \\ 3 & 2 \end{pmatrix} \tag{1.2}$$

For instance, there is a 1 on the far right, second row of the above matrix, meaning that in the graph there is an edge from vertex 1 to vertex 3. This results in the row (1,3) in the transformed matrix seen above.

Here is code to do this computation in **snow**:

```
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)
}
```

What is new here? First, since we desired the output matrix to be in lexicographical order, we needed a way to keep track of the original indices of the rows. So, we added a column for those numbers to **m**:

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

Second, note the use of R's **lapply()** function. Just as **apply()** calls a specified function on each row (or each column) of a matrix, **lapply()** calls a specified function on each element of a list. The output will also be a list.

In our case here, we need to feed the row chunks of **m** into **clusterApply()**, but the latter requires that we do that via a list. We could have done that using a **for** loop, adding row chunks to a list one by one, but it is more compact to use **lapply()**.

In the end, the manager node receives many parts of the new matrix, which must be combined. It's natural to do that with the **rbind()** function, but again we need to overcome the fact that the parts are packaged in an R list. It's handy to use **do.call()** again, though **Reduce()** would have worked too.

Note carefully that although it's natural to use **rbind()** as mentioned in the preceding paragraph, it's not efficient. This is because each call to **rbind()** causes a new matrix to be allocated, a time-consuming action. It would be better to allocate, say, 50 rows at a time, and fill in the rows as we build the matrix. Whenever we would use up all of a matrix, we would start a new one, and then return all the matrices in a list.

1.5.8 Example: Setting Node IDs and Notification of Cluster Size

Recall that in OpenMP there are functions **omp_get_thread_num()** and **omp_get_num_threads()** that report a thread's ID number and the total number of threads. In MPI, the corresponding functions are **MPI_Comm_rank()** and **MPI_Comm_size()**. It would be nice to have such functions (or such functionality) in **snow**. Here is code for that purpose:

```
# 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)
}
```

Yes, R does allow defining a function within a function. Note by the way the use of the superassignment operator, `<<-`, which assigns to the global level.

After this call, any code executed by a worker node can then determine its node number, e.g. in code such as

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

Or, we could send code from the manager to be executed on the workers:

```
> 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
```

In that first case, since **clusterApply()** returns a value, it was printed out. In the second case, the call

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

asks each worker to evaluate the expression **myinfo$id**; **clusterEvalQ()** then returns the results of evaluating the expression at each worker node.

1.5.9 Shutting Down a Cluster

Don't forget to stop your clusters before exiting R, by calling **stopCluster(clustername)**.

1.6 The multicore Package

As the name implies, the **multicore** package is used to exploit the power of multicore machines. This might seem odd: Since **snow** can be used on either a (physical) cluster of machines or on a multicore machine, while **multicore** can only be used on the latter, one might wonder what, if anything, is to be gained by using **multicore**. The answer is that one might gain in performance, as will be explained.

The package's main function, **mclapply()**, is similar in syntax to **snow**'s **clusterApply()**, and similarly parcels tasks out to the various worker nodes.

The worker nodes here, though, are just different processes on the same machine. Say for example you are running **multicore** on a quad-core machine. Calling **mclapply()** will start (a default value of) 4 new invocations of R on your machine, each of which will work on a piece of your application in parallel. Each invocation has exactly the same R variables set up as your original R process did before the call. Thus all the variables are shared initially (note that qualifier), and you as the programmer do not take any special action to transfer variables from the manager node to the worker nodes, quite a contrast to **snow**.

The way all this is accomplished is that **mclapply()** calls your OS' **fork()** function. (It is thus limited to Unix-family OSs, such as Linux and Macs.) The process that is forked is R itself, with one new copy per desired worker node.

The workers thus start with copies of R all sharing whatever variables existed at the

time of the fork (including locals in the function that you had call **mclapply()**). Thus your code does not have to copy these variables to the workers, which automatically have access to them. But note carefully that the variables are shared only initially, and a write to one copy is NOT reflected in the other copies (including the original one).

The copying of the initial values of the variables from the manager node to the worker nodes is done on a **copy-on-write** basis, meaning that data isn't copied to a node until (and unless) the node tries to access that data. The granularity is at the virtual memory page level (Section **??**). Again, the OS handles this, not R.

Thus some physical copying does occur eventually, done by the OS, so **multicore** will not have as much advantage over **snow** as one might think. However, there may be some latency-hiding advantage (Section **??**). It may be the case that not all workers need to access some variable at the same time, so one worker might do so while others are doing actual computation.

Note too that, in contrast to **snow**, in which a cluster is set up once per session and then repeatedly reused at each **snow** function call, with **multicore** the worker R processes are set up again from scratch each time a **multicore** function is called.

1.6.1 Example: Transforming an Adjacency Matrix, multicore Version

Same application as in Section 1.5.7, and indeed the function **tgonchunk()** below is just a modified version of what we had in the **snow** code.

The call

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

applies the function **tgonchunk()** to every element of the vector **starts** (changed to an R list first), with **m1** and **chunksize** serving as additional arguments to **mclapply()**.

```
# 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)
}
```

1.7 Rdsm

My **Rdsm** package can be used as a threads system regardless of whether you are on a NOW or a multicore machine. It is an extension of a similar package I wrote in 2002 for Perl, called PerlDSM. (N. Matloff, PerlDSM: A Distributed Shared Memory System for Perl, *Proceedings of PDPTA 2002*, 2002, 63-68.) The major advantages of **Rdsm** are:

- It uses a shared-memory programming model, which as noted in Section **??**, is commonly considered in the parallel processing community to be clearer than messag-passing.

- It allows full use of R's debugging tools.

Rdsm gives the R programmer a shared memory view, but the objects are not physically shared. Instead, they are stored in a server and accessed through network sockets,[7] thus enabling a threads-like view for R programmers even on NOWs. There is no manager/-worker structure here. All of the R processes execute the same code, as peers.

Shared objects in **Rdsm** can be numerical vectors or matrices, via the classes **dsmv** and **dsmm**, or R lists, using the class **dsml**. Communication with the server in the vector and matrix cases is done in binary form for efficiency, while serialization is used for lists. There is as a built-in variable **myinfo** that gives a process' ID number and the total number of processes, analogous to the information obtained in **Rmpi** from the functions **mpi.comm.rank()** and **mpi.comm.size()**.

To install, again use **install.packages()** as above. There is built-in documentation, but it's best to read through the code **MatMul.R** in the **examples** directory of the **Rdsm** distribution first. It is heavily commented, with the goal of serving as an introduction to the package.

1.7.1 Example: Inversion of Block-Diagonal Matrices

Let's see how the block-diagonal matrix inversion example from Section 1.5.5 can be handled in **Rdsm**.

```
1  # invert a block diagonal matrix m, whose sizes are given in szs; here
       m
2  # is either an Rdsm or bigmemory shared variable; no return
3  # value--inversion is done in-place; it is assumed that there is one
4  # thread for each block
5
6  bdiaginv <- function(bd,szs) {
7     # get number of rows of bd
8     nrdb <- if(class(bd) == "big.matrix") dim(bd)[1] else bd$size[1]
9     rownums <- getrng(nrdb,szs)
10    myid <- myinfo$myid
11    rng <- rownums[myid,1]:rownums[myid,2]
12    bd[rng,rng] <- solve(bd[rng,rng])
13    barr() # barrier
```

[7] Or, **Rdsm** can be used with the **bigmemory** package, as seen in Section 1.7.3.

```
}

# 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
}
```

The parallel work is basically done in four lines:

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

compared to about 11 lines in the **snow** implementation above. This illustrates the power of the shared-memory programming model over message passing.

1.7.2 Example: Web Probe

In the general programming community, one major class of applications, even on a serial platform, is parallel I/O. Since each I/O operation may take a long time (by CPU standards), it makes sense to do them in parallel if possible. **Rdsm** facilitates doing this in R.

The example below repeatedly cycles through a large list of Web sites, taking measurements on the time to access each one. The data are stored in a shared variable **accesstimes**; the **n** most recent access times are stored. Each **Rdsm** process works on one Web site at a time.

An unusual feature here is that one of the processes immediately exits, returning to the R interactive command line. This allows the user to monitor the data that is being

collected. Remember, the shared variables are still accessible to that process. Thus while the other processes are continually adding data to **accesstimes** (and deleted one item for each one added), the user can give commands to the exited process to analyze the data, say with histograms, as the collection progresses.

Note the use of lock/unlock operations here, with the **Rdsm** variables of the same names.

```
# 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]
```

```
29          # 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 }
```

1.7.3 The bigmemory Package

Jay Emerson and Mike Kane developed the **bigmemory** package when I was developing **Rdsm**; neither of us knew about the other.

The **bigmemory** package is not intended to provide a threads environment. Instead, it is used to deal with a hard limit R has: No R object can be larger than $2^{31} - 1$ bytes. This holds even if you have a 64-bit machine with lots of RAM. The **bigmemory** package solves the problem on a multicore machine, by making use of operating system calls to set up shared memory between processes.[8]

In principle, **bigmemory** could be used for threading, but the package includes no infrastructure for this. However, one can use **Rdsm** in conjunction with **bigmemory**, an advantage since the latter is very efficient.

Using **bigmemory** variables in **Rdsm** is quite simple: Instead of calling **cnewdsm()** to create a shared variable, call **newbm()**.

1.8 R with GPUs

The blinding speed of GPUs (for certain problems) is sure to of interest to more and more R users in the coming years.

[8] It can also be used on distributed systems, by exploiting OS services to map memory to files.

As of today, the main vehicle for writing GPU code is CUDA, on NVIDIA graphics cards. CUDA is a slight extension of C.

You may need to write your own CUDA code, in which case you need to use the methods of Section 1.9. But in many cases you can get what you need in ready-made form, via the two main packages for GPU programming with R, **gputools** and **rgpu**. Both deal mainly with linear algebra operations. The remainder of this section will deal with these packages.

1.8.1 Installation

Note that, due to issues involving linking to the CUDA libraries, in the cases of these two packages, you probably will *not* be able to install them by merely calling **install.packages()**. The alternative I recommend works as follows:

- Download the package in **.tar.gz** form.
- Unpack the package, producing a directory that we'll call **x**.
- Let's say you wish to install to **/a/b/c**.
- Modify some files within **x**.
- Then run

```
R CMD INSTALL -l /a/b/c x
```

Details will be shown in the following sections.

1.8.2 The gputools Package

In installing **gputools**, I downloaded the source from the CRAN R repository site, and unpacked as above. I then removed the subcommand

```
-gencode arch=compute_20,code=sm_20
```

from the file **Makefile.in** in the **src** directory. I also made sure that my shell startup file included my CUDA executable and library paths, **/usr/local/cuda/bin** and **/usr/local/cuda/lib**.

I then ran **R CMD INSTALL** as above. I tested it by trying **gpuLm.fit()**, the **gputools** version of R's regular **lm.fit()**.

The package offers various linear algebra routines, such as matrix multiplication, solution of Ax = b (and thus matrix inversion), and singular value decomposition, as well as some computation-intensive operations such as linear/generalize linear model estimation and hierarchical clustering.

Here for instance is how to find the square of a matrix **m**:

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

The **gpuSolve()** function works like the R **solve()**. The call `gpuSolve(a,b)` will solve the linear system ax = b, for a square matrix **a** and vector **b**. If the second argument is missing, then a^{-1} will be returned.

1.8.3 The rgpu Package

In installing **rgpu**, I downloaded the source code from `https://gforge.nbic.nl/frs/?group_id=38` and unpacked as above. I then changed the file **Makefile**, with the modified lines being

```
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
```

The first line was needed to pick up **-lcuda**, as with **gputools**. The second line was needed to acquire the file **cutil.h** in the NVIDIA SDK, which I had installed earlier at the location see above.

For the third line, I made a file **z.c** consisting solely of the line

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

and ran

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

just to see whether the R include file was.

As of May 2010, the routines in **rgpu** are much less extensive than those of **gputools**. However, one very nice feature of **rgpu** is that one can compute matrix expressions without bringing intermediate results back from the device memory to the host memory, which would be a big slowdown. Here for instance is how to compute the square of the matrix **m**, plus itself:

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

1.9 Parallelism Via Calling C from R

Parallel R aims to be faster than ordinary R. But even if that aim is achieved, it's still R, and thus potentially slow.

One must always decide how much effort one is willing to devote to optimization. For the fastest code, we should not write in C, but rather in assembly language. Similarly, one must decide whether to stick purely to R, or go to the faster C. If parallel R gives you the speed you need in your application, fine; if not, though, you should consider writing part of your application in C, with the main part still written in R. You may find that placing the parallelism in the C portion of your code is good enough, while retaining the convenience of R for the rest of your code.

1.9.1 Calling C from R

In C, two-dimensional arrays are stored in row-major order, in contrast to R's column-major order. For instance, if we have a 3x4 array, the element in the second row and second column is element number 5 of the array when viewed linearly, since there are three elements in the first column and this is the second element in the second column. Of course, keep in mind that C subscripts begin at 0, rather than at 1 as with R. In writing your C code to be interfaced to R, you must keep these issues in mind.

All the arguments passed from R to C are received by C as pointers. Note that the C function itself must return `void`. Values that we would ordinarily return must in the R/C context be communicated through the function's arguments, such as `result` in our example below.

1.9.2 Example: Extracting Subdiagonals of a Matrix

As an example, here is C code to extract subdiagonals from a square matrix.[9] The code is in a file **sd.c**:

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

[9] I wish to thank my former graduate assistant, Min-Yu Huang, who wrote an earlier version of this function.

```
7
8  void subdiag(double *m, int *n, int *k, double *result)
9  {
10   int nval = *n, kval = *k;
11   int stride = nval + 1;
12   for (int i = 0, j = kval; i < nval-kval; ++i, j+= stride)
13      result[i] = m[j];
14 }
```

For convenience, you can compile this by rubnning R in a terminal window, which will invoke GCC:

```
1 % R CMD SHLIB sd.c
2 gcc -std=gnu99 -I/usr/share/R/include -fpic -g -O2 -c sd.c -o sd.o
3 gcc -std=gnu99 -shared -o sd.so sd.o -L/usr/lib/R/lib -lR
```

Note that here R showed us exactly what it did in invoking GCC. This allows us to do some customization.

But note that this simply produced a dynamic library, **sd.o**, not an executable program. (On Windows this would presumably be a **.dll** file.) So, how is it executed? The answer is that it is loaded into R, using R's **dyn.load()** function. Here is an example:

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

Note that we needed to allocate space for `result` in our call, in a variable we've named `result`. The value placed in there by our function is seen above to be correct.

1.9.3 Calling C OpenMP Code from R

Since OpenMP is usable from C, that makes it in turn usable from R. (See Chapter **??** for a detailed discussion of OpenMP.)

The code is compiled and then loaded into R as in Section 1.9, though with the additional step of specifying the `-fopenmp` command-line option in both invocations of GCC (which you run by hand, instead of using **R CMD SHLIB**).

1.9.4 Calling CUDA Code from R

The same principles apply here, but one does have to be careful with libraries and the like.

As before, we want to compile not to an executable file, but to a dynamic library file. Here's how, for the C file **mutlinksforr.cu** presented in the next section, the compile command is

```
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
```

The product of this was **meanlinks.so**. I then tested it on R:

```
> 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
```

1.9.5 Example: Mutual Outlinks

We again take as our example the mutual-outlinks example from Section **??**. Here is an R/CUDA version:

```
// 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;
```

```
22    }
23 }
24
25 // proc1pair() processes one pair of Web sites, i.e. one pair of rows
       in
26 // the nxn adjacency matrix m; the number of mutual outlinks is added
       to
27 // tot
28 __global__ void proc1pair(int *m, int *tot, int n)
29 {
30    // find (i,j) pair to assess for mutuality
31    int pair[2];
32    findpair(threadIdx.x,n,pair);
33    int sum=0;
34    // make sure to account for R being column-major order; R's i-th row
35    // is our i-th column here
36    int startrowa = pair[0],
37        startrowb = pair[1];
38    for (int k = 0; k < n; k++)
39       sum += m[startrowa + n*k] * m[startrowb + n*k];
40    atomicAdd(tot,sum);
41 }
42
43 // meanout() is called from R
44 // hm points to the link matrix, nrc to the matrix size, meanmut to the
       output
45 void meanout(int *hm, int *nrc, double *meanmut)
46 {
47     int n = *nrc,msize=n*n*sizeof(int);
48     int *dm, // device matrix
49        htot, // host grand total
50        *dtot; // device grand total
51     cudaMalloc((void **)&dm,msize);
52     cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
53     htot = 0;
54     cudaMalloc((void **)&dtot,sizeof(int));
```

```
55     cudaMemcpy(dtot,&htot,sizeof(int),cudaMemcpyHostToDevice);
56     dim3 dimGrid(1,1);
57     int npairs = n*(n-1)/2;
58     dim3 dimBlock(npairs,1,1);
59     proc1pair<<<dimGrid,dimBlock>>>(dm,dtot,n);
60     cudaThreadSynchronize();
61     cudaMemcpy(&htot,dtot,sizeof(int),cudaMemcpyDeviceToHost);
62     *meanmut = htot/double(npairs);
63     cudaFree(dm);
64     cudaFree(dtot);
65 }
```

The code is hardly optimal. We should, for instance, have more than one thread per block.

1.10 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。

1.10.1 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。

1.10.2 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 [10]。

[10] 译者注：RStudio 中的调试功能已添加

1.10.3 缺少命令行终端的问题

Parallel R packages such as **Rmpi**, **snow**, **foreach** and so on do not set up a terminal for each process, thus making it impossible to use R's debugger on the workers. What then can one do to debug apps for those packages? Let's consider **snow** for concreteness.

First, one should debug the underlying single-worker function, such as **mtl()** in our mutual outlinks example in Section 1.5.6. Here one would set up some artificial values of the arguments, and then use R's ordinary debugging facilities.

This may be sufficient. However, the bug may be in the arguments themselves, or in the way we set them up. Then things get more difficult. It's hard to even print out trace information, e.g. values of variables, since **print()** won't work in the worker processes. The **message()** function may work for some of these packages; if not, you may have to resort to using **cat()** to write to a file.

Rdsm allows full debugging, as there is a separate terminal window for each process.

1.10.4 Debugging C Called from R

For parallel R that is implemented via R calls to C code, producing a dynamically-loaded library as in Section 1.9, debugging is a little more involved. First start R under GDB, then load the library to be debugged. At this point, R's interpreter will be looping, anticipating reading an R command from you. Break the loop by hitting ctrl-c, which will put you back into *GDB's* interpreter. Then set a breakpoint at the C function you want to debug, say **subdiag()** in our example above. Finally, tell GDB to continue, and it will then stop in your function! Here's how your session will look:

```
1 $ R -d gdb
2 GNU gdb 6.8-debian
3 ...
4 (gdb) run
5 Starting program: /usr/lib/R/bin/exec/R
6 ...
7 > dyn.load("sd.so")
```

1.11 本书中的其它 R 语言示例

See these examples (some nonparallel):

in Sections **??**, **??** (nonparallel) and **??** (nonparallel).

- Parallel Jacobi iteration of linear equations, Section **??**.
- Matrix computation of 1-D FFT, Section **??** (can paralleliza using parallel matrix multiplication).
- Parallel computation of 2-D FFT, Section **??**.
- Image smoothing, Section **??**.