并行机器编程

Norm Matloff（著）
加州大学戴维斯分校

寇强（译）
印第安纳大学

GPU、多核、集群

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`

作者尽了最大努力，但书中错误在所难免

目录

第 1 章 Introduction to Parallel R **1**

1.1 Why Is R Featured in This Book? 1

1.2 R and Embarrassing Parallel Problems 2

1.3 Some Parallel R Packages 2

1.4 Installing and Loading the Packages 3

1.5 The R snow Package 3

1.5.1 Usage 4

1.5.2 Example: Matrix-Vector Multiply, Using parApply() 4

1.5.3 Other snow Functions: clusterApply(), clusterCall() Etc. 6

1.5.4 Example: Parallel Sum 8

1.5.5 Example: Inversion of Block-Diagonal Matrices 11

1.5.6 Example: Mutual Outlinks 13

1.5.7 Example: Transforming an Adjacency Matrix 15

1.5.8 Example: Setting Node IDs and Notification of Cluster Size 17

1.5.9 Shutting Down a Cluster 19

1.6 The multicore Package 19

1.6.1 Example: Transforming an Adjacency Matrix, multicore Version . . . 20

1.7 Rdsm 21

1.7.1 Example: Inversion of Block-Diagonal Matrices 22

1.7.2 Example: Web Probe 23

1.7.3 The bigmemory Package 25

1.8 R with GPUs 25

1.8.1 Installation 26

1.8.2 The gputools Package 26

1.8.3 The rgpu Package 27

1.9 Parallelism Via Calling C from R 28

1.9.1 Calling C from R 28

1.9.2 Example: Extracting Subdiagonals of a Matrix 28
1.9.3 Calling C OpenMP Code from R 30
1.9.4 Calling CUDA Code from R 30
1.9.5 Example: Mutual Outlinks 31
1.10 Debugging R Applications 33
1.10.1 Text Editors 33
1.10.2 IDEs 33
1.10.3 The Problem of Lack of a Terminal 34
1.10.4 Debugging C Called from R 34
1.11 Other R Examples in This Book 34

第 1 章　Introduction to Parallel R

1.1　Why Is R Featured in This Book?

Our main language in the remaining chapters of this book will continue to be C/C++, but we will also have a number of R examples. Why feature R?

- R is the most widely-used programming language for statistics and data manipulation. In this era of Big Data, a number of "parallel R" packages have been developed. In particular, the package **parallel** is now a standard part of the R base distribution.
- The widespread usage of R is epitomized by the fact that Google has developed its own internal style guidelines for it.[①] Oracle now includes R with its Big Data Appliance package.
- R will often be very convenient for the purpose of illustrating various parallel algorithms. This convenience arises from the fact that R has built-in vector and matrix types, as well as a complex number type.

Python also has various parallelization libraries, notably **multiprocessing**. The topic of parallel Python in Chapter **??**.

Examples in this chapter will be kept simple. But parallel R can be applied to parallelize very large, complex problems.

There is a 5-minute tutorial in Appendix **??** of this book. In reading that material, keep in mind the role of R lists.

The key to parallel R—list manipulation:

Many parallel R packages depend heavily on R lists. Both input and output arguments often take the form of lists, as do the return values. The reader may wish to review the material on R lists in Appendix **??**.

[①]I personally do not like those guidelines, preferring my own, but the mere fact that Google has set up its own guidelines shows the significance they place on R.

1.2 R and Embarrassing Parallel Problems

It should be noted that parallel R packages tend to work well only on embarrassingly parallel problems. Recall that this was defined (in Sec. **??**) to mean algorithms that are not only easily parallelized but that also have relatively small communication needs.[2] As we know, it is often the case that only embarrassingly parallel algorithms give good performance, but is especially so in the case of R, for the following reasons.

The functional programming nature of R implies that technically, any write to a single vector or matrix element, say

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

means that the entire vector or matrix is rewritten.[3] There are exceptions to this (probably more and more in each new version of R), but generally we must accept that parallel vector and matrix code will be expensive in R.[4]

For non-embarrassingly parallel applications, one should consider interfacing R to parallel C code, which is described in Section 1.9.

1.3 Some Parallel R Packages

Here are a few parallel R packages:

- Message-passing or scatter/gather (Section **??**): **Rmpi**, **snow**, **foreach**, **rmr** (cloud), **Rhipe** (cloud), **multicore**[5], **rzmq**
- Shared-memory: **Rdsm**, **bigmemory**
- GPU: **gputools**, **rgpu**

A far more extensive list is at `http://cran.r-project.org/web/views/HighPerformanceComputing.html`.

Starting with version 2.14, the R base distribution includes a **parallel** package, consisting of **snow** and **multicore**. (Earlier versions must download these separately.) For this reason, both of these are covered here. I will also cover **Rdsm/bigmemory** and **gputools**.

[2] This latter condition eliminates many iterative algorithms, even though they may be easily parallelized.
[3] Element assignment in R *is* a function call, with arguments in the above case being **x**, 3 and 8.
[4] R's new Reference classes may change this somewhat.
[5] The **multicore** package runs on multicore, i.e. shared memory, machines, but does not share data in the read/write sense.

1.4 Installing and Loading the Packages

Installing:

As noted, if you have R 2.14 or later, you already have **snow** and **multicore**. In general, with the exception of **Rgpu**, all of the packages above are downloadable/installable from CRAN, the official R repository for contributed code, at `http://cran.r-project.org`. Here's what to do, say for **snow**:

Suppose you want to install in the directory **/a/b/c/**. The easiest way to do so is use R's **install.packages()** function, say:

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

This will install **snow** in the directory **/a/b/c/snow**.

You'll need to arrange for the directory **/a/b/c** (not **/a/b/c/snow**) to be added to your R library search path. I recommend placing a line

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

in a file **.Rprofile** in your home directory (this is an R startup file).

In some cases, due to issues such as locations of libraries, you may need to install a CRAN package "by hand." See Section 1.8.1 and 1.8.3 below.

Loading a package:

Call **library()** to load a package. For instance, to load **parallel**, type:

```
> library(parallel)
```

1.5 The R snow Package

The real virtue of **snow** is its simplicity. The concept is simple, the implementation is simple, and very little can go wrong. Accordingly, it may be the most popular type of parallel R in use today.

The **snow** package runs directly via network sockets (probably the most common usage, since the user does not have to install anything besides **snow**), or on top of **Rmpi** (R's interface to MPI), PVM or NWS.

It operates under a scatter/gather model (Sec. **??**).

For instance, just as the ordinary R function **apply()** applies the same function to all rows of a matrix (example below), the **snow** function **parApply()** does that in parallel, across multiple machines; different machines will work on different rows. (Instead of running on several machines, we might one several **snow** clients on one multicore machine.)

1.5.1 Usage

After loading **snow**, by typing

```
> library(snow)
```

one sets up a **snow** cluster, by calling the **snow** function **makeCluster()**. The named argument **type** of that function indicates the networking platform, e.g. “MPI” or “SOCK.” The last indicates that you wish **snow** to run on TCP/IP sockets that it creates itself, rather than going through MPI etc.

In the examples here, I used “SOCK,” on machines named **pc48** and **pc49**, setting up the cluster this way:[6]

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

Note that the above R code sets up **worker nodes** at the machines named **pc48** and **pc49**; these are in addition to the **manager node**, which is the machine on which that R code is executed.

By the way, if you want to make worker nodes on the same machine as the manager (typically on a multicore machine), use **localhost** as the machine name.

There are various other optional arguments. One you may find useful is **outfile**, which records the result of the call in the file **outfile**. This can be helpful for debugging if the call fails.

1.5.2 Example: Matrix-Vector Multiply, Using parApply()

To introduce **snow**, consider a simple example of multiplication of a vector by a matrix. We set up a test matrix:

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

[6] If you are on a shared-file system group of machines, try to stick to ones for which the path to R is the same for all, to avoid problems.

```
[4,] 4 10
[5,] 5 11
[6,] 6 12
```

We will multiply the vector $(1,1)^T$ (T meaning transpose) by our matrix **a**. In this small example, of course, we would do that directly:

```
> a %*% c(1,1)
     [,1]
[1,] 8
[2,] 10
[3,] 12
[4,] 14
[5,] 16
[6,] 18
```

But let's see how we could do it using R's **apply()** function, still in serial form, as it will set the stage for extending to parallel computation.

R's **apply()** function calls a user-specified, scalar-valued function to each of the rows (or each of the columns) of a user-specified matrix. This returns a vector. To use **apply()** for our matrix-times-vector problem here, define a dot product function:

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

Now call **apply()**:

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

This call applies the function **dot()** to each row (indicated by the 1, with 2 meaning column instead of row) of the matrix **a**; a row plays the role of the first argument to **dot()**, and with c(1,1) playing the role of the second argument. In other words, the first call to **dot()** will be

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

The **snow** library function **parApply()** then extends **apply()** to parallel computation. Let's use it to parallelize our matrix multiplication, across our the machines in our cluster **cls**:

```
> parApply(cls,a,1,dot,c(1,1))
```

```
[1] 8 10 12 14 16 18
```

What **parApply()** did was to send some of the rows of the matrix to each node, also sending them the function **dot()** and the argument **c(1,1)**. Each node applied **dot()** to each of the rows it was given, and then returned the results to be assembled by the manager node.

R's **apply()** function is normally used in scalar-valued situations, meaning that **f()** in a call **apply(m,i,f)** is scalar-valued. If **f()** is vector-valued, then a matrix will be returned instead of a vector, with each column of that matrix being the result of calling **f()** on a row or column of **m**. The same holds for **parApply()**.

1.5.3 Other snow Functions: clusterApply(), clusterCall() Etc.

In the last section we introduced the **parApply()** function. Its general call form is

- **parApply():**
  The call

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

results in the rows of the matrix **m** parceled out to the various worker nodes of **cls**, at which **f()** will be applied to each row, with optional parameters in the position indicated by the ellipsis. The argument **DIM** is 1 for row operations, 2 for columns.

The return value is a vector (or possibly a matrix, as noted above).

The virtue of **snow** is its simplicity. Thus it does not have a lot of complex functions. But there is certainly more than just **parApply()**. Here are a few more:

- **clusterApply():**
  This function may be the most heavily-used function in **snow**. The call

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

  runs **f()** at each worker node in **cls**. Here **individualargs** is an R list (if it is a vector, it will be converted to a list). When **f()** is called at node i of the cluster, its arguments will be as follows. The first argument will be the ith element of the **individualargs**, i.e. **individualargs**[[**i**]]. If arguments indicated by the ellipsis are in the call (optional), then these will be passed to **f()** as its second, third and so on arguments.

  If **individualargs** has more elements than the number of nodes in the cluster, then **cls** will be recycled (treating it as a vector), so that most or all nodes will call **f()** on more than element of **individualargs**.

The return value is an R list, whose ith component is the result of the call to **f()** on the ith element of **individualargs**.

Typically the list **individualargs** consists of the work to be split up and done in parallel.

- **clusterApplyLB():**

  This is a load-balancing form of **clusterApply()**, aimed at addressing the performance issues we discussed on Chapter **??**.

  To explain the difference between the two forms of the cluster-apply operation, suppose our cluster consists of 10 nodes, and that we have 25 tasks for them to do (i.e. **individualargs** has length 25). With **clusterApply()**, the following will occur:

  – The first 10 tasks will be sent to the workers, one task per worker.

  – The manager will wait for all 10 tasks to complete, and then will send out the next 10.

  – The manager will wait for these 10 tasks to complete, and then will send out the remaining 5.

  – The manager will wait for the results of these 5, and then return the 25 results to the caller.

  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 11th 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 ith 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 ith 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 Example: Parallel Sum

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 ith 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) {
   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)
}
```

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

```
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:

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

$b
[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 list-s. 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**.

```r
# 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
```

[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]
         # add to accesstimes, in sliding-window fashion
         lock("acclock")
         if (naccesstimes[1] < ww) {
            naccesstimes[1] <- naccesstimes[1] + 1
            accesstimes[naccesstimes[1]] <- acc
         } else {
            # out with the oldest, in with the newest
            newvec <- c(accesstimes[-1],acc)
            accesstimes[] <- newvec
         }
         unlock("acclock")
      }
   }
}
```

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 **instal-l.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/lo-cal/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**:

```
// 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
```

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

```

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];
}
```

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

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

```
> 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
```

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;
   }
}

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

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

1.10 Debugging R Applications

The built-in debugging facilities in R are primitive, but alternatives are available.

1.10.1 Text Editors

However, if you are a Vim editor fan, I've developed a tool that greatly enhances the power of R's debugger. Download **edtdbg** from R's CRAN repository. It's also available for Emacs.

Vitalie Spinu's `ess-tracebug` runs under Emacs. It was modeled roughly on `edtdbg`, but has more Emacs-specific features than does `edtdbg`.

1.10.2 IDEs

I'm personally not a fan of IDEs, but some excellent ones are available.

REvolution Analytics, a firm that offers R consulting and develops souped-up versions of R, offers an IDE for R that includes nice debugging facilities. It is only available on Windows, and even then only for those who have Microsoft Visual Studio.

The developers of StatET, a platform-independent Eclipse-based IDE for R added a debugging tool in May 2011.

The people developing RStudio, another a platform-independent IDE for R, also plan to begin work on a debugger, beginning summer 2011.

1.10.3 The Problem of Lack of a Terminal

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:

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

1.11 Other R Examples in This Book

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 **??**.