What a difference! Now that **x** and **y** are shared by the processes, we can access them directly, making our code vastly simpler.

Note carefully that we are talking about human efficiency here, not machine efficiency. Use of shared memory can greatly simplify our code, with far less clutter, so that we can write and debug our program much faster than we could in a message-passing environment. That doesn't mean our program itself has faster execution speed.

It will turn out, though, that **Rdsm** can indeed enjoy a speed advantage over other parallel R libraries for some applications. We'll return to this issue in Section 5.5.

5.3 High-Level Introduction to Shared-Memory Programming: Rdsm Library

Though one sometimes needs to write directly in C/C++ in order to truly maximize speed, it is highly desirable to stay within R whenever possible, in order to leverage R's powerful data manipulation and statistical operations. This is the philosophy underlying R libraries such as **Rmpi** and **snow**.

However, those are message-passing approaches, and as mentioned above, the inherent simplicity of the shared-memory programming paradigm makes it highly desirable to have a shared-memory parallel computation library for R. At the time of this writing, my package **Rdsm** is the only such library. You can download it from the R contributed package repository, CRAN.

R itself is not threaded (or more accurately, R does not make threading available at the R programming level). But **Rdsm** brings threads programming to R. In addition to **Rdsm**'s direct value, it is also useful here in this chapter as a gentle introduction to shared-memory programming. The fact that R does the heavy lifting in terms of data and statistical operations means we can focus on learning shared-memory coding.

Rdsm version 2.0.0 is used here, as it has an easy user interface. Ironically, the shared-memory library **Rdsm** uses the message-passing software **snow** for some infrastructure.

5.3.1 Use of Shared Memory

Modern operating systems allow the programmer to request that a chunk of memory be made available on a shared basis by any process that holds a certain key. The **bigmemory** library in R's CRAN code repository enables this for R programmers. **Rdsm** builds on this.

Specifically, the **Rdsm** programmer makes a certain call to set up each shared variable, and **snow** is used to distribute the associated keys to the **Rdsm** threads, thus enabling the threads to share variables!

The shared variables must take the form of matrices, a **bigmemory** constraint. Note that one must use brackets in referencing them. For instance, to print the shared matrix, write

```
print(m[,])
```

rather than

```
print(m)
```

As will be seen below, **snow** is also used as the mechanism to launch the threads

Though **Rdsm** is intended to run on shared-memory machines, **bigmemory** allows shared storage in the form of files. Thus **Rdsm** can also be used to provide the shard-memory world view on a distributed system, e.g. clusters.

5.4 Example: Matrix Multiplication

The standard "Hello World" example of the parallel processing community is matrix multiplication. Here is the **Rdsm** code, along with a small test.

5.4.1 The Code

```
1 # matrix multiplication; the product u %*% v is computed
2 # on cls, and stored in w; w is a big.matrix object
3
4 mmulthread <- function(u,v,w) {
5     require(parallel)
6     myidxs <- splitIndices(nrow(u),myinfo$nwrkrs)[[myinfo$id]]
```

```
7     w[myidxs,] <- u[myidxs,] %*% v[,]
8     0  # don't do expensive return of result
9  }
10
11 test <- function(cls) {
12    mgrinit(cls)
13    mgrmakevar(cls,"a",6,2)
14    mgrmakevar(cls,"b",2,6)
15    mgrmakevar(cls,"c",6,6)
16    a[,] <- 1:12
17    b[,] <- rep(1,12)
18    clusterExport(cls,"mmulthread")
19    clusterEvalQ(cls,mmulthread(a,b,c))
20    print(c[,])
21 }
```

Here is a test run:

```
> library(Rdsm)
> c2 <- shmcls(2)
> source("~/R/Rdsm/examples/MMul.R")
> test(c2)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    8    8    8    8    8    8
[2,]   10   10   10   10   10   10
[3,]   12   12   12   12   12   12
[4,]   14   14   14   14   14   14
[5,]   16   16   16   16   16   16
[6,]   18   18   18   18   18   18
```

Here we first set up a two-node **snow** cluster **c2** (remember, this is not necessarily a physical cluster), by calling the **Rdsm** convenience function **shmcls()**. The code **test()** is run on the **snow** manager node.

First, **Rdsm**'s **mgrinit()** is called to initialize the **Rdsm** system, after which we set up three matrices in shared memory, **a**, **b** and **c** (**a** and **b** could have been nonshared). This action will distribute the necessary keys to the **snow** worker nodes.

Then **snow**'s **clusterEvalQ()** is used to launch the threads.[4] On a quad-core machine running four **Rdsm** threads, for example, **mmulthread()** will run on all threads at once (though it probably won't be the case that all threads are running the same line of code simultaneously).

[4] Another example of *remote procedure call*, mentioned in Section 4.3.6.

Now, how does **mmulthread()** work? The basic idea is break the rows of the argument matrix **u** into chunks, and have each thread work on one chunk.[5] Say there are 1000 rows, and we have a quadcore machine (on which we've set up a four-node **snow** cluster). Thread 1 would handle rows 1-250, thread 2 would work on rows 251-500 and so on. The chunks are assigned in the code

```
myidxs <- splitIndices(nrow(u),myinfo$nwrkrs)[[myinfo$id]]
```

calling the **snow** function **splitIndices()**. For example, the value of **myidxs** at thread 2 will be 251:500. The built-in **Rdsm** variable **myinfo** is an R list containing t**nwrkrs**, the total number of threads, and **id**, the ID number of the thread. On thread 2 in our example here, those numbers will be 4 and 2, respectively.

The reader should note the "me, my" point of view that is key to threads programming. Remember, each of the threads is (more or less) simultaneously executing **mmulthread()**. So, the code in that function must be written from the point of view of a particular thread. That's why we put the "my" in the variable name **myidxs**. We're writing the code from the anthropomorphic view of imagining ourselves executing the code as a particular thread. That thread is "me," and so the row indices are "my" indices, hence the name **myidxs**.

Each thread multiplies **v** by the thread's own chunk of **u**, placing the result in the corresponding chunk of **w**:

```
w[myidxs,] <- u[myidxs,] %*% v[,]
```

Here we are using the property of multiplying partitioned matrices, explained in Section A.7.

This last line of code is like our y <−x back in Section 5.2. Unlike a message-passing approach, we had no shipping of object back and forth among threads; the objects are "already there," and we access them simply and directly.

In this small example, the simplicity of shared-memory programming occurs only in this one line of code. But in a complex program, the increase in simplicity, readability and so on would be quite substantial. Furthermore, since copying can really slow down a program, the reduction in copying due to using the shared-memory paradigm can mean substantial speed increases.

[5] Some parallel algorithms partition both **u** and **v**. See Chapter 10.

Incidentally, the shared-memory nature of our code is also reflected in the fact that our result, the matrix **w**, is not returned to the caller. Instead, it is simply available as a shared variable to all parties who hold the key for that variable.

Indeed, we can access that variable (**c**, the actual argument corresponding to **w** after our call to **mmulthread()**) back at the **snow** manager:

```
> c[,]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    8    8    8    8    8    8
[2,]   10   10   10   10   10   10
[3,]   12   12   12   12   12   12
[4,]   14   14   14   14   14   14
[5,]   16   16   16   16   16   16
[6,]   18   18   18   18   18   18
```

In fact, the **Rdsm** package includes instructions for saving a key to a file and then loading it from another invocation of R on the same machine. The latter will then be able to access the shared variable as well.

5.4.2 Timing Comparison

We won't do extensive timing experiments here, but let's just check that the code is indeed providing a speedup:

```
> n <- 5000
> m <- matrix(runif(n^2),ncol=n); system.time(m %*% m)
   user  system elapsed
345.077   0.220 346.356
> cls <- shmcls(4)
> mgrinit(cls)
> mgrmakevar(cls,"msh",n,n)
> mgrmakevar(cls,"msh2",n,n)
> msh[,] <- m
> clusterExport(cls,"mmulthread")
> system.time(clusterEvalQ(cls,mmulthread(msh,msh,msh2)))
   user  system elapsed
  0.004   0.000  91.863
```

So, a fourfold increase in the number of cores yielded almost a fourfold increase in speed, very good.

5.4.3 Leveraging R

It was pointed out earlier that a good reason for avoiding C/C++ if possible is to be able to leverage R's powerful built-in operations. In this example, we made use of R's built-in matrix-multiply capability, in addition to its ability to extra subsets of matrices.

This is a common strategy. To solve a big problem, we break it into smaller ones of the same type, apply R's tools to the small problems, and then somehow combine to obtain the final result. This of course is a general parallel processing design pattern, but with a difference in that we need to find appropriate R tools. R is an interpreted language, thus with a tendency to be slow, but its basic operations typically make use of functions that are written in C, which are fast. Matrix multiplication is such an operation, so our approach here does work well.

5.5 Shared Memory Can Bring A Performance Advantage

In addition to the tendency of shared-memory code to be clearer and more concise, in many applications we can reap a significantly performance gain as well. Message-passing systems by definition do a lot of copying of data, sometimes very large amounts of data, that is often unnecessary. With shared memory, we can read and write our needed data directly, as you'll see concretely below.

Note first, though, that shared-memory access may involve hidden data copying. Each cache coherency transaction involves copying of data, and if such transactions occur frequently, it can add up to large amounts. Indeed, some of *that* copying may be unnecessary, say when a cache block is brought in but never used much afterward. Thus shared-memory programming is not necessarily a "win," but it will become clear below that it can be much faster for some applications, relative to other R parallel libraries such as **snow**, **multicore**, **foreach** and even **Rmpi**.

To see why, here is a version of **mmulthread()** using the **snow** library:

```
snowmmul <- function(cls,u,v) {
    require(parallel)
    idxs <- splitIndices(nrow(u),length(cls))
    mmulchunk <- function(idxchunk) u[idxchunk,] %*% v
    res <- clusterApply(cls,idxs,mmulchunk)
```

n	# cores	**Rdsm** time	**Snow** time
2000	8	4.640	6.398
3000	16	10.892	18.010
3000	24	8.778	19.001

Table 5.1: Rdsm vs. snow

```
    Reduce(rbind,res)
}
```

This test code was used:

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

It turns out to be considerably slower than the **Rdsm** implementation, as seen in Table 5.1.

The results are for various sizes of nxn matrices, and various numbers of cores.

One of the culprits is the line

```
Reduce(rbind,res)
```

in the **snow** version. This involves a lot of copying of data, and possibly worse, multiple allocation of large matrices, greatly sapping speed. This is in stark contrast to the **Rdsm** case, in which the threads directly wrote their chunked-multiplication results to the desired output matrix. Note that the **Reduce()** operation itself is done serially, and though we might

try to parallelize that too, that itself would require lots of copying, and thus may be difficult to make work well.

This of course was not a problem especially with **snow**. The same **Reduce()** operation or equivalent would be needed with **multicore**, **foreach** (using the **.combine** option), **Rmpi** and so on.[6] **Rdsm**, by writing the results directly to the desired output, avoids that problem.

It is clear that there are many applications with similar situations, in which tools like **snow** etc. do a lot of serial data manipulation following the parallel phase. In addition, iterative algorithms, such as k-means clustering (Section 5.8) involve repeated alternating between a serial and parallel phase. **Rdsm** should typically give faster speed than do the others in these applications.

We should not overlook **Rmpi**. Its **mpi.gather()** and **mpi.gatherv()** functions deposit items directly into their ultimate intended destination, as we saw in Chapter 4. But we would still need to spend time copying the two multiplicands to the workers.

The shared-memory vs. message-passing debate is a long-running one in the parallel processing community. It has been traditional to argue that the shared-memory paradigm doesn't scale well (Section 2.5), but the advent of modern multicore systems, especially GPUs, has done much to counter that argument.

5.6 Locks and Barriers

These are two central concepts in shared-memory programming.

5.6.1 Race Conditions and Critical Sections

Consider software to manage online airline reservations, and for simplicity, assume there is no overbooking. At some point in the program, there will be a section consisting of one or more lines of code whose purpose is to perform the actual reservation of a seat: The customer's name and other data are entered into the database for the given flight on the given day. That section of code is known as a *critical section*, for the following reason.

Imagine a scenario in which two customers who want the given flight on the given day log in to the reservation system at about the same time. Each of

[6] With **multicore**, we would have a little less copying, as explained in Section 3.4.1.

them will be running a separate thread of the program (though of course they won't be aware of this). Suppose only one seat is left on the flight. It could happen that each thread finds that there is a seat remaining on the flight, and thus each thread enters the critical section—and thus each thread books its customer for the flight! One of the threads will be slightly ahead of the other, and the later thread will overwrite what the earlier one wrote. In other words, the first customer thinks she has successfully booked the flight, but actually has not.

Now you can see why such a section of code is called "critical." It is fraught with danger, with the situation being known as a *race condition*. Handling this problem properly is called *synchronization*. (Sorry, you will be bombarded with terminology in the next few paragraphs.)

Also, we say that the problem with the flight reservations above stemmed from a failure to update the reservation records *atomically*. The Greek word *atom* means "indivisible," and the allusion here is that trouble may arise if we "divide" the read (checking for availability of a seat) and write (committing the seat to the customer) phases in the critical section, as opposed to doing both phases in one indivisible action. Doing that atomically would mean that a thread does the read and write as an indivisible pair, without having any other thread being able to act between the two phases.

5.6.2 Locks

What we need to avoid race conditions is a mechanism that will limit access to the critical section to only one thread at a time, i.e. *mutual exclusion*. A common mechanism is a *lock variable* or *mutex*. Most thread systems include functions **lock()** and **unlock()**, applied to a lock variable. Just before a critical section, one inserts a call to **lock()**, execution of which will work as follows.

Supppose the lock variable is already locked, due to some other thread currently being inside the critical section. Then the thread making the call to **lock()** will *block*, meaning that it will just freeze up for the time being, not returning yet. When the thread currently in the critical section finally exits, it will call **unlock()**, and the blocked thread will now unblock: This thread will enter the critical section, and relock the lock, so that any other thread trying to get in will block.

To make this concrete, consider this toy example, in **Rdsm**. We've initialized **Rdsm** as a two-thread system, **c2**, and set up a 1x1 shared variable **tot**.

```
# this function is not reliable; if 2 threads both try to
# increment the total at about the same time, they could
# interfere with each other

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

library(parallel)
clusterExport(c2,"s")
tot[1,1] <- 0
clusterEvalQ(c2,s(1000))
tot[1,1]  # should be 2000, but likely far from it
```

I did two runs of this. On the first one, the final value of **tot[1,1]** was 1021, while the second time it was 1017. Neither time did it come out 2000 as it "should." Moreover, the result was random.

The problem here is that the action

```
tot[1,1] <- tot[1,1] + 1
```

is not atomic. We could have the following sequence of events:

```
thread 1 reads tot[1,1], finds it to be 227
thread 2 reads tot[1,1], finds it to be 227
thread 1 writes 228 to tot[1,1]
thread 2 writes 228 to tot[1,1]
```

Here, **tot[1,1]** should be 229, but is only 228.

But with locks, everything works fine:

```
# here is the reliable version, surrounding the
# increment by lock and unlock, so only 1 thread
# can execute it at once
s1 <- function(n) {
   for (i in 1:n) {
      realrdsmlock("totlock")
      tot[1,1] <- tot[1,1] + 1
      realrdsmunlock("totlock")
```

```
    }
}

mgrmakelock(c2,"totlock")

tot[1,1] <- 0
clusterExport(c2,"s1")
clusterEvalQ(c2,s1(1000))
tot[1,1]  # will print out 2000, the correct number
```

5.6.3 Barriers

Another key structure is that of a *barrier*, which is used to synchronize all the threads. Suppose for instance that we need one thread to perform some special action, but that we need to have the other threads wait for that action to be performed. The threads system will provide a function to call that accomplishes this. In **Rdsm**, this function is named **barr()**, and when a thread calls it, the thread will block until all threads have called it. Afterward, they all proceed to the next line of code.

Note that internally a barrier needs to be implemented with a lock. You, the application programmer, won't see the lock (unless you're curious), but you do need to be aware that it is there, as locks affect performance. Which brings us to the next section...

5.6.4 Lockfree Synchronization

Bear in mind that locks and barriers are "necessary evils." We do need them to ensure correct execution of our program, but they slow things down. For instance, we say that lock variables *serialize* a program in the section they are used, i.e. they change its parallel character to serial. And contention for locks can cause lots of cache coherency transactions, definitely putting a damper on performance. Thus one should always try to find clever ways to avoid locks and barriers if possible.

One way to do this is to take advantage of the hardware. Modern processors typically include a variety of hardware assists to make synchronization more efficient.

For example, Intel machines allow a machine instruction to be prefixed by a special byte called a **lock** prefix. It orders the hardware to lock up the

system bus while the given instruction is executing—so that the execution is atomic. (The fact that this prefix, a hardware operation, is named **lock** should not be confused with lock variables in software.)

Under the critical section approach, code to do an atomic add to **y** would look something like this:

```
lock the lock
add operand to y
unlock the lock
```

By contrast, we could do all this with a single machine instruction:

```
lock add %edx, tot
```

if say the instruction uses the EDX register.

The OpenMP threaded programming framework, which we'll discuss later in this chapter, includes a keyword named **atomic**. It instructs the compiler to try to find a hardware construct like the above to implement mutual exclusion, rather than taking the less efficient critical section route. Details later.

Also, the C++ Standard Template Library contains related constructs, such as the function **fetch_add()**, which again instructs the compiler to attempt to find an atomic hardware solution to the update-total example above.

5.7 Example: Transformation of an Adjacency Matrix

Say we have a graph with an adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{5.1}$$

For example, there is an edge from vertex 1 to vertex 2, but not one from vertex 3 to 1. We'd like to transform this to a two-column matrix that

displays the links, in this case

$$\begin{pmatrix} 1 & 2 \\ 2 & 1 \\ 2 & 4 \\ 3 & 2 \\ 3 & 4 \\ 4 & 1 \\ 4 & 2 \\ 4 & 3 \end{pmatrix} \tag{5.2}$$

For instance, the (4,3) in the last row means there is an edge from vertex 4 to 3, corresponding to the 1 in row 4, column 3 of the adjacency matrix.

5.7.1 The Code

Here is **Rdsm** code for this:

```
1  # inputs a graph adjacency matrix, and outputs a two-column matrix
2  # listing the edges emanating from each node
3
4  library(Rdsm)
5
6  # arguments:
7  #    a:  adjacency matrix
8  #    lnks:  edges matrix; shared, nrow(a)^2 rows and 2 columns
9  #    counts:  numbers of edges found by each thread; shared
10
11 # in this version, the matrix lnks must be created ahead of time; since
12 # the number of rows is uknown a priori, one must allow for the worst
13 # case, nrow(a)^2 rows; after the run, the number of actual rows will be
14 # in counts[1,length(cls)]
15
16 getlinksthread <- function(a,lnks,counts) {
17    require(parallel)
18    nr <- nrow(a)
19    # get my assigned portion of a
20    myidxs <- getidxs(nr)
21    myout <- apply(a[myidxs,],1,function(rw) which(rw==1))
22    # myout[[i]] now lists the edges from node myidxs[1] + i - 1
23    nmyedges <- Reduce(sum,lapply(myout,length))  # my total edges
24    me <- myinfo$id
```

```
25      counts[1,me] <- nmyedges
26      barr()
27      if (me == 1) {
28         # use cumsum() to determine where each node will store its results
29         # in lnks
30         counts[1,] <- cumsum(counts[1,])
31      }
32      barr()
33      # lnksidx will be the next row to write within lnks
34      lnksidx <- if (me == 1) 1 else counts[1,me-1] + 1
35      for (idx in myidxs) {
36         # corresponding index to idx within myout
37         jdx <- idx - myidxs[1] + 1
38         myoj <- myout[[jdx]]
39         endwrite <- lnksidx + length(myoj) - 1
40         if (!is.null(myoj)) {
41            lnks[lnksidx:endwrite,] <- cbind(idx,myoj)
42         }
43         lnksidx <- endwrite + 1
44      }
45      0  # don't do expensive return of result
46  }
47
48  test <- function(cls) {
49      mgrinit(cls)
50      mgrmakevar(cls,"x",6,6)
51      mgrmakevar(cls,"lnks",36,2)
52      mgrmakevar(cls,"counts",1,length(cls))
53      x[,] <- matrix(sample(0:1,36,replace=T),ncol=6)
54      clusterExport(cls,"getlinksthread")
55      clusterEvalQ(cls,getlinksthread(x,lnks,counts))
56      print(lnks[1,counts[1,length(cls)],])
57  }
```

Here we will have our first example of the use of barriers.

As with the earlier examples, the division of labor involves assigning different chunks of rows of the adjacency matrix to different **Rdsm** threads. (There definitely *are* other design patterns in the parallel processing world. See Section **??**.) To determine the chunks, we could call **snow**'s **splitIndices()** as before (as shown in the commented-out code), but actually **Rdsm** provides a simpler wrapper for that, **getidxs()**, which we've called here:

```
myidxs <- getidxs(nr)
```

```
myout <- apply(a[myidxs,],1,function(rw) which(rw==1))
```

The matrix **myout** will now give a row-by-row listing of the column numbers of all the 1s in the rows of this thread's chunk. Remember, our ultimate output matrix, **lnks**, will have one row for each such 1, so the information in **myout** will be quite useful.

Indeed, this thread will compute its portion of **lnks**, and then place it there. But in order to do so, this thread must know where in **lnks** to start writing. And for that, this thread needs to know how many 1s were found by threads prior to it. If for instance thread 1 finds eight 1s and thread 2 finds three, then thread 3 must start writing at row 8 + 3 + 1 = 12 in **lnks**. Thus we need to find the overall 1s counts (across all rows of a thread) for each thread,

```
nmyedges <- Reduce(sum,lapply(myout,length))  # my total edges
```

and then need to find cumulative sums, and share them. To do this, we'll have thread 1 find those sums, and place them in our shared variable **counts**:

```
me <- myinfo$id
counts[1,me] <- nmyedges
barr()
if (me == 1) {
   counts[1,] <- cumsum(counts[1,])
}
barr()
```

Note the barrier calls just before and just after thread 1 does this work. The first call is needed because thread 1 can't start finding the cumulative sums before the individual counts are ready. Then we need the second barrier, because all the threads will be using use of the cumulative sums, and we need to be sure they're ready before such use.

Now that our thread knows where in **lnks** to write its results, it can go ahead:

```
lnksidx <- if (me == 1) 1 else counts[1,me-1] + 1
for (idx in myidxs) {
   jdx <- idx - myidxs[1] + 1
   myoj <- myout[[jdx]]
   endwrite <- lnksidx + length(myoj) - 1
   if (!is.null(myoj)) {
      lnks[lnksidx:endwrite,] <- cbind(idx,myoj)
```

```
    }
    lnksidx <- endwrite + 1
}
```

The loop handles each row in this thread's chunk, converting row numbers in **myidxs** to those in the original adjacency matrix.

5.7.2 Overallocation of Memory

A problem is having to allocate the **lnks** matrix to handle the worst case, thus wasting space and execution time. The problem is that we don't know in advance the size of our "output," in this case the argument **lnks**. In our little example above, the adjacency matrix was of size 4x4, while the edges matrix was 7x2. We know the number of columns will be 2, but the number of rows is unknown *a priori*.

The above version of our code solves the problem simply by insisting that the user allow for the worst case. For an nxn adjacency matrix, the edges matrix could have as many as n^2 rows. If n is large, this could be a major performance issue.

Before discussing an alternate approach, note that the user can determine the number of "real" rows in **lnks** by inspecting **counts[1,length(cls)]** after the call returns, as seen in the test code.

One approach would be to postpone allocation until we know how big the **lnks** matrix needs to be, which we will know after the cumulative sums are calculated. We could have thread 1 then create the shared matrix **lnks**, by calling **bigmemory** directly rather than using **mgrmakevar()**. To distribute the shared-memory key for this matrix, thread 1 would save the **bigmemory** descriptor to a file, then have the other threads get access to **lnks** by loading from the file.

Actually, this problem is common in parallel processing applications. We will return to it in Section 5.14.2.

5.7.3 Timing Experiment

For comparison, here is a serial version of the code:

```
1 > getlinksnonpar
2 function(a,lnks) {
3     nr <- nrow(a)
```

```
4     myout <- apply(a[,],1,function(rw) which(rw==1))
5     nmyedges <- Reduce(sum,lapply(myout,length))
6     lnksidx <- 1
7     for (idx in 1:nr) {
8        jdx <- idx
9        myoj <- myout[[jdx]]
10       endwrite <- lnksidx + length(myoj) - 1
11       if (!is.null(myoj)) {
12          lnks[lnksidx:endwrite,] <- cbind(idx,myoj)
13       }
14       lnksidx <- endwrite + 1
15    }
16    0
17 }
```

```
> n <- 10000
> system.time(getlinksnonpar(x,lnks))
   user  system elapsed
 26.170   1.224  27.516
```

(For convenience, we are still using **Rdsm** to set up the shared variables, though we run in non-**Rdsm** code.)

Now try the parallel version:

```
> cls <- shmcls(4)
> mgrinit(cls)
> mgrmakevar(cls,"counts",1,length(cls))
> mgrmakevar(cls,"x",n,n)
> mgrmakevar(cls,"lnks",n^2,2)
> x[,] <- matrix(sample(0:1,n^2,replace=T),ncol=n)
> clusterExport(cls,"getlinksthread")
> system.time(clusterEvalQ(cls,getlinksthread(x,lnks,counts)))
   user  system elapsed
  0.000   0.000   7.783
```

So, the parallel code did indeed speed things up.

5.8 Example: K-Means Clustering

In discussion of parallel computation for data science, an example application almost as common as matrix multiplication is k-means clustering.

The goal is to form k groups from our data matrix, hopefully in a way that makes visual (or other) sense. Let's see how that can be implemented in **Rdsm**.

The general k-means method itself is quite simple, using an iterative algorithm. At any step during the iteration process, the k groups are summarized by their centroids.[7] We iterate the following:

1. For each data point, i.e. each row of our data matrix, determine which centroid this point is closest to.
2. Add this data point to the group corresponding to that centroid.
3. After all data points are processed in this manner, update the centroids to reflect the current group memberships.
4. Next iteration.

This example will bring in a concept in shared-memory work that didn't arise in our matrix multiplication example, related to the phrase, "After all data points are processed..." in step 3. Some other new concepts will come up as well, all to be explained below.

5.8.1 The Code

So, here is the code, again with a small test function:

```
1  # k-means clustering on the data matrix x, with k clusters and ni
2  # iterations; final cluster centroids placed in cntrds
3
4  # initial centroids taken to be k randomly chosen rows of x; if a
5  # cluster becomes empty, its new centroid will be a random row of
6  # x
7
8  library(Rdsm)
9
10 # arguments:
11 #    x:  data matrix x; shared
12 #    k:  number of clusters
13 #    ni:  number of iterations
14 #    cntrds:  centroids matrix; row i is centroid i; shared, k by ncol(x)
```

[7] If we have m variables, then the centroid of a group is the m-element vector of means of those variables within this group.

```
15 #    cinit:  optional initial values for the centroids; k by ncol(x)
16 #    sums:  scratch matrix; sums[j,] contains the count
17 #       and sum for cluster j; shared, k by 1+ncol(x)
18 #    lck:  lock variable; shared
19
20 kmeans <- function(x,k,ni,cntrds,sums,lck,cinit=NULL) {
21    require(parallel)
22    require(pdist)
23    nx <- nrow(x)
24    # get my assigned portion of x
25    # myidxs <- splitIndices(nx,myinfo$nwrkrs)[[myinfo$id]]
26    myidxs <- getidxs(nx)
27    myx <- x[myidxs,]
28    # random initial centroids if none specified
29    if (is.null(cinit)) {
30       if (myinfo$id == 1)
31          cntrds[,] <- x[sample(1:nx,k,replace=F),]
32       barr()
33    } else cntrds[,] <- cinit
34
35    # mysum()sums the rows in myx corresponding to the indices idxs; we
36    # also produce a count of those rows
37    mysum <- function(idxs,myx) {
38       c(length(idxs),colSums(myx[idxs,,drop=F]))
39    }
40    for (i in 1:ni) {  # ni iterations
41       # node 1 is sometimes asked to do some "housekeeping"
42       if (myinfo$id == 1) {
43          sums[] <- 0
44       }
45       barr()  # other nodes wait for node 1 to do its work
46       # find distances from my rows of x to the centroids, then
47       # find which centroid is closest to each such row
48       dsts <- matrix(pdist(myx,cntrds[,])@dist,ncol=nrow(myx))
49       nrst <- apply(dsts,2,which.min)
50       # nrst[i] contains the index of the nearest centroid to row i in
51       # myx
52       tmp <- tapply(1:nrow(myx),nrst,mysum,myx)
53       # in the above, we gather the observations in myx whose closest
54       # centroid is centroid j, and find their sum, placing it in
55       # tmp[j]; the latter will also have the count of such observations
56       # in its leading component
57       # next, we need to add that to sums[j,], as an atomic operation
```

```
58          realrdsmlock(lck)
59          # the j values in tmp will be strings, so convert
60          for (j in as.integer(names(tmp))) {
61             sums[j,] <- sums[j,] + tmp[[j]]
62          }
63          realrdsmunlock(lck)
64          barr()  # wait from sums[,] to be ready
65          if (myinfo$id == 1) {
66             # update centroids, using a random data point if a cluster
67             # becomes empty
68             for (j in 1:k) {
69               # update centroid for cluster j
70               if (sums[j,1] > 0) {
71                  cntrds[j,] <- sums[j,-1] / sums[j,1]
72                } else cntrds[j] <<- x[sample(1:nx,1),]
73             }
74          }
75       }
76       0  # don't do expensive return of result
77    }
78
79    test <- function(cls) {
80       library(parallel)
81       mgrinit(cls)
82       mgrmakevar(cls,"x",6,2)
83       mgrmakevar(cls,"cntrds",2,2)
84       mgrmakevar(cls,"sms",2,3)
85       mgrmakelock(cls,"lck")
86       x[,] <- matrix(sample(1:20,12),ncol=2)
87       clusterExport(cls,"kmeans")
88       clusterEvalQ(cls,kmeans(x,2,1,cntrds,sms,"lck",
89          cinit=rbind(c(5,5),c(15,15))))
90    }
91
92    test1 <- function(cls) {
93       mgrinit(cls)
94       mgrmakevar(cls,"x",10000,3)
95       mgrmakevar(cls,"cntrds",3,3)
96       mgrmakevar(cls,"sms",3,4)
97       mgrmakelock(cls,"lck")
98       x[,] <- matrix(rnorm(30000),ncol=3)
99       ri <- sample(1:10000,3000)
100      x[ri,1] <- x[ri,1] + 5
```

```
101     ri <- sample(1:10000,3000)
102     x[ri,2] <- x[ri,2] + 5
103     clusterExport(cls,"kmeans")
104     clusterEvalQ(cls,kmeans(x,3,50,cntrds,sms,"lck"))
105 }
```

Let's first discuss the arguments of **kmeans()**. Our data matrix is **x**, which is described in the comments as a shared variable (on the assumption that it will often be such) but actually need not be.

By contrast, **cntrds** needs to be shared, as the threads repeatedly use it as the iterations progress. We have thread 1 writing to this variable,

```
if (myinfo$id == 1) {
   for (j in 1:k) {
      if (sums[j,1] > 0) {
         cntrds[j,] <<- sums[j,-1] / sums[j,1]
       } else cntrds[j] <<- x[sample(1:nx,1),]
   }
}
```

at the end of each iteration, and all threads reading it:

```
dsts <- matrix(pdist(myx,cntrds[,])@dist,ncol=nrow(myx))
```

If **cntrds** were not shared, the whole thing would fall apart. When thread 1 would write to it, it would become a local variable for that thread, and the new value would not become visible to the other threads. Note that as in our previous examples, we store our function's final result, in this case **cntrds**, in a shared variable, rather than as a return value.

The argument **sums** is also shared by necessity. It is only used to store intermediate results, but again this variable is written to by some threads and subsequently read by others, hence must be shared.

Another argument to **kmeans()** that is shared is **lck**, a lock variable, to be discussed below.

So, let's look at the actual code, starting with

```
# get my assigned portion of x
# myidxs <- splitIndices(nx,myinfo$nwrkrs)[[myinfo$id]]
myidxs <- getidxs(nx)
myx <- x[myidxs,]
```

Once again our approach will be to break the data matrix into chunks of rows. Each thread will handle one chunk, finding distances from rows in its chunk to the current centroids. How is the above code preparing for this?

Note again the "me, my" point of view here, pointed out in Section 5.4 and present in almost any threads function. The code here is written from the point of view of a particular thread. So, the code first needs to determine this thread's rows chunk.

Why have this separate variable, **myx**? Why not just use **x[myidxs,]**? First, having the separate variable results in less cluttered code. But secondly, repeated access to **x** could cause a lot of costly cache misses and cache coherency actions.

Next we see another use of barriers:

```
if (is.null(cinit)) {
   if (myinfo$id == 1)
      cntrds[,] <- x[sample(1:nx,k,replace=F),]
   barr()
} else cntrds[,] <- cinit
```

We've set things up so that if the user does not specify the initial values of the centroids, they will be set to k random rows of **x**. We've written the code so that thread 1 performs this task, but we need the other threads to wait until the task is done. If we didn't do that, one thread might race ahead and start accessing **cntrds** before it is ready. Our call to **barr()** ensures that this won't happen.

We have a similar use of a barrier at the beginning of the main loop:

```
if (myinfo$id == 1) {
   sums[] <- 0
}
barr()  # other nodes wait for node 1 to do its work
```

We need to compute the distances to the various centroids from all the rows in this thread's portion of our data:

```
dsts <- matrix(pdist(myx,cntrds[,])@dist,ncol=nrow(myx))
```

R's **pdist** library comes to the rescue! This package, which we saw in Section 3.6, finds all distances from the rows of one matrix to the rows of another, exactly what we need. So, here again, we are leveraging R! (Indeed, an alternate way to parallelize the computation from what we are

doing here would be to parallelize **pdist()**, say using **Rdsm** instead of **snow** as before.)

Next, we leverage R's **which.min()** function, which finds indices of minima (not the minima themselves). We use this to determine the new group memberships for the data points in **myx**:

```
nrst <- apply(dsts,2,which.min)
# nrst[i] contains the index of the nearest centroid to row i in
# myx
```

Next, we need to collect the information in **nrst** into a more usable form, in which we have, for each centroid, a vector stating the indices of all rows in **myx** that now will belong to that centroid's group. For each centroid, we'll also need to sum all such rows, in preparation for later averaging them to find the new centroids.

Again, we can leverage R to do this quite compactly (albeit needing a bit of thought):

```
mysum <- function(idxs,myx) {
   c(length(idxs),colSums(myx[idxs,,drop=F]))
}
...
tmp <- tapply(1:nrow(myx),nrst,mysum,myx)
```

But remember, all the threads are doing this! For instance, thread 1 is finding the sum of its rows that are now closest to centroid 6, but thread 4 is doing the same. For centroid 6, we will need the sum of all such rows, across all such threads.

In other words, multiple threads may be writing to the same row of **sums** at about the same time. Race condition ahead! So, we need a lock:

```
lock(lck)
for (j in names(tmp)) {
   j <- as.integer(j)
   sums[j,] <- sums[j,] + tmp[[j]]
}
unlock(lck)
```

The **for** loop here is a critical section. Without the restriction, chaos could result. Say for example two threads want to add 3 and 8 to a certain total, respectively, and that the current total is 29. What could happen is that they both see the 29, and compute 32 and 37, respectively, and then write those numbers back to the shared total. The result might be that the new

total is either 32 or 37, when it actually should be 40. The locks prevent such a calamity.

A refinement would be to set up k locks, one for each row of **sums**. As noted earlier, locks sap performance, by temporarily serializing the execution of the threads. Having k locks instead of one might ameliorate the problem here.

After all the threads are done with this work, we can have thread 1 compute the new averages, i.e. the new centroids. But the key word in the last sentence is "after." We can't let thread 1 do that computation until we are sure that all the threads are done. This calls for using a barrier:

```
barr()
if (myinfo$id == 1) {
   for (j in 1:k) {
     if (sums[j,1] > 0) {
        cntrds[j,] <<- sums[j,-1] / sums[j,1]
      } else cntrds[j] <<- x[sample(1:nx,1),]
   }
}
```

As noted earlier, the shared variable **sums** serves as storage for intermediate results, not only sums of the data points in a group, but also their counts. We can now use that information to compute the new centroids:

```
if (myinfo$id == 1) {
   for (j in 1:k) {
     # update centroid for cluster j
     if (sums[j,1] > 0) {
        cntrds[j,] <- sums[j,-1] / sums[j,1]
      } else cntrds[j] <<- x[sample(1:nx,1),]
   }
}
```

5.8.2 Timing Experiment

Let n denote the number of rows in our data matrix. With k clusters, we have to compute nk distances per iteration, and then take n minima. So the time complexity is $O(nk)$.

This is not very promising for parallelization. In many cases $O(n)$ (fixing k here) does not provide enough computation to overcome overhead issues.

However, with our code here, there really isn't much overhead. We copy the data matrix just once,

```
myx <- x[myidxs,]
```

and thus avoid problems of contention for shared memory and so on.

It appears that we can indeed get a speedup from our parallel version some cases:

```
> x <- matrix(runif(100000*25),ncol=25)
> system.time(kmeans(x,10))  # kmeans() function in base R, k = 10
   user  system elapsed
  8.972   0.056   9.051
> cls <- shmcls(4)
> mgrinit(cls)
> mgrmakevar(cls,"cntrds",10,25)
> mgrmakevar(cls,"sms",10,26)
> clusterExport(cls,"kmeans")
> mgrmakevar(cls,"x",100000,25)
> x[,] <- x
> system.time(clusterEvalQ(cls,kmeans(x,10,10,cntrds,sms,lck)))
   user  system elapsed
  0.000   0.000   4.086
```

A bit more than 2X speedup for four cores, fairly good in view of the above considerations.

5.9 Example: Bucket Sort with Sampling

Here we develop an **Rdsm** version of our earlier **Rmpi** example. Again, the strategy to sort a vector **x** using r threads is quite simple:

- Take a small sample from **x** to obtain estimates of its quantiles.
- Use the quantiles to partition the real number line into r intervals, such that approximately the same number of elements of **x** will fall into each interval. (This will give us load balance.)
- Each thread finds the elements of **x** that fall into its interval.
- Each thread applies R's **sort()** function to its own data.
- Each thread places its sorted numbers in the proper place in **x**.

The vector **x** will be sorted in-place.

5.9.1 The Code

```
1  # bucket sort with sampling
2
3  # vector x is broken into chunks according to cut points; this implies
4  # that all the numbers in chunks i are <= all those in chunk j > i; thus
5  # each chunk can be sorted and then placed into its proper place in x
6
7  # the cuts are obtained by first sorting a sample of x and then
8  # computing r - 1 quantiles, where r is the number of threads
9
10 # arguments:
11 #
12 #    x:  vector to be sorted; shared; sorted in place
13 #    counts:  intermediate result; shared, length = length(cls)
14 #    samp:  intermediate result; shared; length = nsamp
15 #    nsamp:  number of elements of x to sample
16
17 bsort <- function(x,counts,samp,nsamp=1000) {
18    me <- myinfo$id
19    # make local copy of x to avoid cache coherency overhead
20    tmpx <- x[1,]
21    if (me == 1) {  # sample to get quantiles
22       samp[1,] <- sort(tmpx[sample(1:length(tmpx),nsamp,replace=F)])
23    }
24    barr()
25    # determine my interval
26    r <- myinfo$nwrkrs
27    k <- floor(nsamp / r)
28    if (me > 1) mylo <- samp[1,(me-1) * k]
29    if (me < r) myhi <- samp[1,me * k]
30    # get my chunk and sort it
31    if (me == 1) myx <- tmpx[tmpx <= myhi] else
32       if (me == r) myx <- tmpx[tmpx > mylo] else
33          myx <- tmpx[ tmpx > mylo & tmpx <= myhi]
34    myx <- sort(myx)
35    # need to decide where in x to place myx
36    lx <- length(myx)
37    counts[1,me] <- lx
38    barr()
```

```
39      if (me == 1) counts[1,] <- cumsum(counts[1,])
40      barr()
41      # place my sorted chunk back in x
42      if (me == 1) x[1,1:lx] <- myx else {
43          start <- counts[1,me-1] + 1
44          fin <- start + lx - 1
45          x[1,start:fin] <- myx
46      }
47  }
48
49  test <- function(cls) {
50      mgrinit(cls)
51      mgrmakevar(cls,"a",1,25)
52      mgrmakevar(cls,"counts",1,length(cls))
53      mgrmakevar(cls,"smp",1,10)
54      a[1,] <- runif(25)
55      print(a[1,])
56      clusterExport(cls,"bsort")
57      clusterEvalQ(cls,bsort(a,counts,smp,nsamp=10))
58      print(a[1,])
59  }
```

So, let's see how the code works, starting with this:

```
tmpx <- x[1,]
```

Here each thread makes its own local copy of **x**. This seems innocuous, but it is quite important. If all the threads were to repeatedly access **x** itself, there would be lots of cache coherency actions, sapping performance. Indeed, I ran a version of the code that did not make a local copy, and found it to be about 20% slower.

Next, we do the sampling:

```
if (me == 1) {  # sample to get quantiles
   samp[1,] <- sort(tmpx[sample(1:length(tmpx),nsamp,replace=F)])
}
barr()
```

Again, a barrier is needed to ensure that **samp** is ready before the other threads start using it.

We then leverage R's subsetting capabilities to determine which elements of **x** fall into this thread's interval, and then sort those elements:

```
r <- myinfo$nwrkrs
k <- floor(nsamp / r)
if (me > 1) mylo <- samp[1,(me-1) * k]
if (me < r) myhi <- samp[1,me * k]
if (me == 1) myx <- tmpx[tmpx <= myhi] else
   if (me == r) myx <- tmpx[tmpx > mylo] else
      myx <- tmpx[ tmpx > mylo & tmpx <= myhi]
myx <- sort(myx)
```

Next, we have a pattern similar to that seen in the example in Section 5.7.1. There, each thread worked on a set of items whose size became known only during the midst of execution. This information was necessary in order to know where a thread needed to write its results back to a certain shared vector, **lnks**. We have the same situation here; each thread needs to know where in **x** to write the sorted chunk that the thread has computed. As with the previous example, this is done by having each thread report its chunk size in a shared vector, **counts**, and then computing cumulative sums:

```
lx <- length(myx)
counts[1,me] <- lx
barr()
if (me == 1) counts[1,] <- cumsum(counts[1,])
barr()
if (me == 1) x[1,1:lx] <- myx else {
   start <- counts[1,me-1] + 1
   fin <- start + lx - 1
   x[1,start:fin] <- myx
}
```

5.9.2 Timing Experiment

Serial sorting of n numbers has an optimal time complexity of $O(n \log n)$. Since the logarithm function grows slowly with n (in fact, the larger n is, the flatter the curve at that point), a complexity of $O(n \log n)$ is not that much larger than $O(n)$. For that reason, gaining speed via parallelism may be an uphill battle.

In such situations, overhead issues can be paramount. As mentioned earlier, our making a local copy of **x**

```
tmpx <- x[1,]
```

can significantly reduce overhead in the form of cache coherency actions. Nevertheless, even the above line may generate some such actions, and in

any case, for large n that is a lot of copying, which itself takes time.

To investigate, I ran simulations, using the code

```
test1 <- function(cls,n) {
    mgrinit(cls)
    mgrmakevar(cls,"a",1,n)
    mgrmakevar(cls,"counts",1,length(cls))
    mgrmakevar(cls,"smp",1,1000)
    a[1,] <- runif(n)
    clusterExport(cls,"bsort")
    print(system.time(clusterEvalQ(cls,bsort(a,counts,smp,nsamp=1000))))
}
```

For the case labeled "r = 1" below, i.e. single-thread, I simply ran R's **sort()**:

```
z <- runif(100000000); system.time(sort(z))
```

n	r	time
25000000	1	9.841
25000000	4	10.604
25000000	8	9.055
25000000	16	9.061
100000000	1	44.417
100000000	4	40.013
100000000	8	34.378
100000000	16	34.136

With n = 25000000, even using 16 cores gives us only a slight performance gain, if any.[8] Using 4 cores actually is worse than using just 1.

With the larger problem n = 100000000, we receive only about a 9% speedup from running 4 threads, and even with 8 threads the speedup is only around 23%. Moreover, going to 16 threads doesn't seem to help any further.

There are of course other sorting algorithms we could try, and if we really need the speed, we should probably switch to C/C++. But the example does show that not all problems parallelize well

[8] All these numbers are subject to some statistical variation.

5.10 OpenMP

The standard method for programming directly on multicore machines, is to use threads libraries, which are available for all modern operating systems. On Unix-family systems, for example, the **pthreads** library is quite popular.

The programmer then calls functions in the threads library, such as the **pthread_mutex_lock()** function in the **pthreads** library to lock a lock variable. However, this can become very tedious, so higher-level libraries were developed specifically with parallel computation in mind, such as OpenMP, Threads Building Blocks and Cilk++. Here we introduce OpenMP.

An OpenMP application still uses threads, but at a higher level of abstraction. One accesses OpenMP through C, C++ or FORTRAN. R users can write an OpenMP application in one of those languages, and then call the application from R, using either the **.C()** or **.Call()** functions available in R for that purpose. To keep things simple, we will stick just to C and**.C()** here. (In order to facilitate interface with R, though, we have used **double** type instead of **float**.)

5.11 Example: Finding the Maximal Burst in a Time Series

Consider a time series of length n. We may be interested in bursts, periods in which a high average value is sustained. Of course, a period of just two or three time points, say, would be too short to count, so we might stipulate that we look only at periods of at least k consecutive points. So, we wish to find the period of at least k consecutive time points that has the maximal mean value.

The time complexity of this application is $O(n(n-k)$, which for fixed k and varying n is the same as $O(n^2)$. This growth rate in n suggests that this is a good candidate for parallelization.

5.11.1 The Code

```
1  // OpenMP example program, Burst.c; burst() finds period of highest
2  // burst of activity in a time series
3
4  #include <omp.h>
```