Rdsm Overview

Norm Matloff

Rdsm is a library for doing parallel processing in R. It uses the *shared-memory paradigm*, which has two advantages:

- It generally involves simpler code, which makes it easier to write and read.
- In some cases it can bring a significant advantage in execution speed.

While it runs best on multicore machines, it can also run on a collection of computers that share a common file system.

Rdsm's infrastructure is built on the **bigmemory** and **synchronicity** libraries for access to shared memory (or common files), and uses the **snow** library (actually part of R's **parallel** library) for launching threads.

This document will introduce **Rdsm**.

1 Clarity and Conciseness of Shared-Memory Programming

On a multicore machine, the standard method for parallel programming is to have a program run in multiple instantiations called *threads*, which run in parallel, one per core. The key point is that all the threads hold the program's global variables in common, the *shared-memory programming paradigm*, or *threaded programming*. **Rdsm** brings this paradigm to R.

The shared-memory programming world view is considered by many in the parallel processing community to be one of the clearest forms of parallel programming.[1] Let's see why.

Suppose for instance we wish to copy **x** to **y**. In a message-passing setting such as **Rmpi**, **x** and **y** may reside in processes 2 and 5, say. The programmer might write code like

```
mpi.send.Robj(x,tag=0,dest=5)
```

to run on process 2, and write code

```
y <- mpi.recv.Robj(tag=0,source=2)
```

[1] See Chandra, Rohit (2001), *Parallel Programming in OpenMP*, Kaufmann, pp.10ff (especially Table 1.1), and Hess, Matthias *et al* (2003), Experiences Using OpenMP Based on Compiler Directive Software DSM on a PC Cluster, in *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools*, Michael Voss (ed.), Springer, p.216.

to run on process 5. By contrast, in a shared-memory environment, the variables **x** and **y** would be shared, and the programmer would merely write for process 5

```
y <- x
```

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. But that doesn't mean our program itself has faster execution speed. However, as shown later in this document, **Rdsm** can indeed bring a significant performance advantage in some applications.

2 Example: Matrix Multiplication

The standard "Hello World" example of the parallel processing community is matrix multiplication. Here is code from the **Rdsm** distribution **examples/** directory:

2.1 The Code

```
1  # matrix multiplication; the product u %*% v is computed, and
2  # stored in w
3
4  # each thread executes this
5  mmul <- function(u,v,w) {
6     require(parallel)
7     # determine which rows of u this thread will handle
8     myidxs <- splitIndices(nrow(u),myinfo$nwrkrs)[[myinfo$id]]
9     # multiply v by this thread's rows in u
10    w[myidxs,] <- u[myidxs,] %*% v[,]
11    # done
12    invisible(0)  # don't do expensive return of result
13 }
14
15 test <- function(cls) {
16    # set up
17    require(parallel)
18    # initialize Rdsm
19    mgrinit(cls)
20    # create the shared variables, and given them values
21    mgrmakevar(cls,"a",6,2)
22    mgrmakevar(cls,"b",2,6)
23    mgrmakevar(cls,"c",6,6)
24    a[,] <- 1:12
```

```
25      b[,] <- rep(1,12)
26      # give the code to the threads
27      clusterExport(cls,"mmul")
28      # here is the actual program execution
29      clusterEvalQ(cls,mmul(a,b,c))
30      print(c[,])
31  }
```

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**, by calling an **Rdsm** convenience function **shmcls()**. The code **test()** is run on the manager node, meaning the one from which we set up the cluster.

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**. This action will distribute the necessary **bigmemory** keys to the **snow** worker nodes.

Snow's **clusterEvalQ()** is used to launch the threads. Suppose here and below that we are on a quadcore machine running four **Rdsm** threads. Then **mmul()** will run on all four threads/cores at once (though it probably won't be the case that all threads are running the same line of code simultaneously).

Now, how does **mmul()** work? The basic idea is break the rows of the argument matrix **u** into chunks, and have each thread work on one chunk. 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 **nwrkrs**, the total number of threads, and **id**, the ID number of this 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 **mmmul()**. 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 the code being executed by 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[,]
```

This last line of code is like our y <− x back in Section 1. Unlike a message-passing approach, we had no shipping of objects 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.

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 **bigmemory** key for that variable.

Indeed, we can access that variable (**c**, the actual argument corresponding to **w** after our call to **mmul()**) back at the 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, **Rdsm** includes utilities **saveshvar()** and **loadshvar()** 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.

2.2 Speedup over Serial Code

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,"mmul")
> system.time(clusterEvalQ(cls,mmul(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. Of course, this is a classic embarrassingly parallel application, so we should expect good speedup, but it's good to confirm it.

Moreover, we'll later next that **Rdsm** is much faster on this particular application than is another R parallel processing library.

2.3 A Possible Performance Advantage for Rdsm

In Section 1, we had a small example comparing the shared-memory and message-passing paradigms. Shared-memory was much clearer and more concise, great for "human efficiency."

But don't let the fact that our shared-memory code had no send/receive operations deceive you into thinking the code necessarily also has faster execution. True, send/receive ops can be slow, especially if they involve voluminous copying (say if **x** is a large matrix). However, the innocuous-looking operation y <− x may involve hidden copying too, due to *cache coherency* actions, and thus is not necessarily a "win."

In this particular application, though, shared-memory does indeed give us a major performance advantage. Here is a version of **mmul()** using the **multicore** library (also part of the **parallel** library):

```
# mc.cores is the number of cores to use in computation
mmul1 <- function(u,v,mc.cores) {
    require(parallel)
    idxs <- splitIndices(nrow(u),mc.cores)
    res <- mclapply(idxs,function(idxchunk) u[idxchunk,] %*% v)
    Reduce(rbind,res)
}
```

It turns out to be considerably slower than the **Rdsm** implementation:

```
> system.time(mmul1(m,m,4))
   user  system elapsed
186.555   1.264 188.886
```

This is about double the run time achieved by **Rdsm**.

The culprit is the line

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

in the **multicore** version, which involves a lot of copying of data, 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.

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 to large systems, but the advent of modern multicore systems, especially GPUs, has done much to counter that argument.

3 Barriers and Locks

Two major tools in shared-memory programming are *barriers* and *locks*, which in **Rdsm** involve the functions **barr()**, **lock()** and **unlock()**:[2]

- When a thread calls **barr()**, the thread will *block*, i.e. not return to the caller, until *all* of the threads have made this call.
- When a thread makes the call **lock(l)**, for some lock variable **l**, the result will depend on whether **l** is currently locked or unlocked:
  - If **l** is locked, the calling thread will be blocked until **l** becomes unlocked, at which time the call will return and the thread will proceed,[3] and the lock will be relocked.
  - If **l** is unlocked, the calling thread will return immediately, and the lock will be relocked.

Why are these constructs central to shared-memory programming? Again, some code from the **examples/** directory will illustrate the ideas.

In the file **BSort.R**, for instance, we are doing *bucket sort with sampling*. The details of how this sort works are not important here (see the comments in the code), but the point is that we must first draw a random sample from the given array. We have thread 1 do this, placing the sample in the shared variable **samp**, while the others wait:

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

The action described in the phrase "while the others wait" is implemented in the call to **barr()**. Remember, the threads are running simultaneously, and we must ensure that no thread attempts to access **samp** before it is ready; the barrier achieves that goal for us.

To see why/how locks are used, consider the file **KMeans.R**. This is a famous clustering method, but again the details are not important here. What is important is that there is a shared matrix **sums**, to which all the threads are adding numbers. Here is the relevant excerpt of the code:

```
lock(lck)
# the j values in tmp will be strings, so convert
for (j in as.integer(names(tmp))) {
   sums[j,] <- sums[j,] + tmp[[j]]
}
unlock(lck)
```

The **for** loop here is known as a *critical section*, meaning that we need to ensure that only one thread is executing in that section of code at a time. Without that restriction, chaos could result. Say for example

[2] The latter two come from the **synchronicity** library.

[3] If several threads had been waiting at the time of the call, only one will unblock, and the rest will continue waiting.

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**. Locks sap performance, by temporarily serializing the execution of the threads. Having k locks instead of one might ameliorate the problem here.