Introduction to the `doRedis` Package

Bryan W. Lewis
blewis@illposed.net

January 30, 2011

1 Introduction

The `doRedis` package provides a parallel back end for `foreach` using Redis and the corresponding `rredis` package. It lets users easily run parallel jobs across multiple R sessions.

Steve Weston's `foreach` package is a remarkable parallel computing framework for the R language. Similarly to lapply-like functions, foreach maps functions to data and aggregates results. Even better, foreach lets you do this in parallel across multiple CPU cores and computers. And even better yet, foreach abstracts the parallel computing details away into modular back end code. Code written using foreach works sequentially in the absence of a parallel back end, and works uniformly across different back ends, allowing programmers to write code largely independent of specific parallel implementations. The `foreach` package has many other wonderful features outlined in its package documentation.

Redis is a fast, persistent, networked database with many innovative features, among them a blocking stack-like data structure (Redis "lists"). This feature makes Redis useful as a lightweight back end for parallel computing. The `rredis` package provides a native R interface to Redis used by `doRedis`.

1.1 Why doRedis?

Why write a `doRedis` package? After all, the `foreach` package already has available many parallel back end packages, including `doMC`, `doSNOW` and `doMPI`.

The `doRedis` package allows for dynamic pools of workers. New workers may be added at any time, even in the middle of running computations. This feature is relevant, for example, to modern cloud computing environments. Users can make an economic decision to "turn on" more computing resources at any time in order to accelerate running computations. Similarly, modern

cluster resource allocation systems can dynamically schedule R workers as cluster resources become available.

The doRedis package makes it particularly easy to run parallel jobs across different operating systems. Although doRedis was developed on GNU/Linux systems, it works equally well on other Unix systems like Mac OS X and even on Windows systems. Back end parallel R worker processes are effectively anonymous–they may run anywhere as long as all the R package dependencies required by the task at hand are available.

Unlike the doSNOW back end, doRedis can aggregate results incrementally, significantly reducing required memory overhead for problems that return large data.

2 Obtaining and Installing the Redis server

Redis is an open source project available from http://redis.io, with development versions and source code available from Github at http://github.com/antirez/redis/tarball/1.3.6 and http://github.com/antirez/redis. A Windows version of Redis is available from: http://github.com/dmajkic/redis.

It is not necessary to "install" Redis to use it. One may download the code, compile it, and run it in place. We include an example command-line procedure applicable to most POSIX operating systems for completeness.

```
wget http://github.com/antirez/redis/tarball/1.3.6
tar xf antirez-redis-1.3.6*.tar.gz
cd antirez-redis-<<version>>
make
# <<Some output from your C compiler>>
```

At this point, unless an error occurred, you have a working copy of Redis. The Redis server is completely configured by the file redis.conf. In order to run the Redis server as a background process, edit this file and change the lines:

```
daemonize no
timeout 300
```

to:

```
daemonize yes
timeout 0
```

You may wish to peruse the rest of the configuration file and experiment with the other server settings as well. Finally, start up the Redis server with

```
./redis-server ./redis.conf
```

2.1 Supported Platforms

The Redis server is written in ANSI C and supported on most POSIX systems including GNU/Linux, Solaris, *BSD, and Mac OS X. A MinGW version is available for Windows systems.

The doRedis package for R is available for all R platforms.

3 doRedis Examples

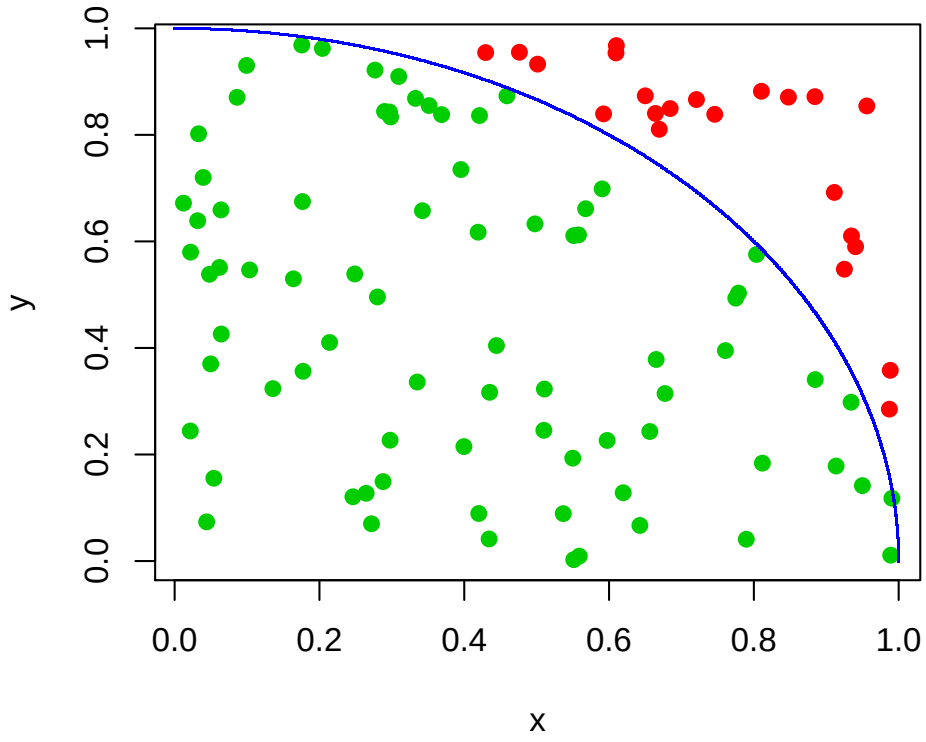
We explore operation of many `doRedis` features through a few examples. Unless otherwise noted, we assume that Redis is installed and running on the local machine ("localhost") as outlined in Section 2 above.

3.1 A Simple Example

The simple example below is one version of a Monte Carlo approximation of π. Variations on this example are often used to illustrate parallel programming ideas.

Listing˜1: Monte Carlo Example

```
> library('doRedis')
> registerDoRedis('jobs')
> startLocalWorkers(n=2, queue='jobs')
> foreach(icount(10),.combine=sum,.multicombine=TRUE,.inorder=FALSE) %dopar%
        4*sum((runif(1000000)^2 + runif(1000000)^2)<1)/10000000
[1] 3.144212
> removeQueue('jobs')
```



The figure illustrates how the method works. We randomly choose points in the unit square. The ratio of points that lie inside the arc of the unit circle (green) to the total number of points provides an approximation of the area of 1/4 the area of the unit circle–that is, an approximation of $\pi/4$. Each one of the 10 iterations of the loop computes a scaled approximation using 1,000,000 such points. We then sum up each of the 10 results to get an approximation of π using all 10,000,000 points.

The `doRedis` package uses the idea of a "work queue" to dole out jobs to available resources. A set of jobs are placed in the queue which are then consumed by workers. The line

```
registerDoRedis('jobs')
```

registers the `doRedis` back end with `foreach` using the user-specified work queue name "jobs" (you are free to use any name you wish for the work queue).

The next line:

```
startLocalWorkers(n=2, queue='jobs')
```

starts up two worker R sessions on the local machine, both listening for work on the queue "jobs." The worker sessions don't display any output by default. The `startLocalWorkers` function can instruct the workers to log messages to output files or stdout if desired.

You can verify that workers are in fact waiting for work from the "jobs" queue with:

```
getDoParWorkers()
```

which should return 2, for the two workers we just started. Note that the number of workers may change over time (unlike most other parallel back ends for `foreach`). The `getDoParWorkers` function returns the current number of workers in the pool. Note that the number returned should only be considered to be an estimate of the actual number of available workers.

The next lines actually run our Monte Carlo code:

```
foreach(icount(10),.combine=sum,.multicombine=TRUE,.inorder=FALSE) %dopar%
      4*sum((runif(1000000)^2 + runif(1000000)^2)<1)/10000000
```

This parallel loop consists of 10 iterations (tasks) using the `icount` iterator function. (It's also possible to use more traditional loop variables in `foreach` loops.) We specify that the results from each task should be passed to the `sum` function with `.combine=sum`. Setting the `.multicombine` option to `TRUE` tells `foreach` that the `.combine` function accepts an arbitrary number of function arguments (some aggregation functions only work on two arguments). The `.inorder=FALSE` option tells foreach that results may be passed to the `.combine` function as they arrive, in any order. The `%dopar%` operator instructs `foreach` to use the `doRedis` back end that we previously registered to place each task in the work queue. Finally, each iteration runs the scaled estimation of π using 1,000,000 points.

3.2 Dynamic Worker Pools and Heterogeneous Workers

It's pretty simple to run parallel jobs across computers with `doRedis`, even if the computers have heterogeneous operating systems (as long as one of them is running a Redis server). It's also very straightforward to add more parallel workers during a running computation. We do both in this section.

We'll use the simple bootstrapping example from the `foreach` documentation to illustrate the ideas of this section. The results presented here were run on the following systems:

- A GNU/Linux dual-core Opteron workstation, host name *master*.
- A Windows Server 2003 quad-core Opteron system.

We installed R version 2.11.0 (2010-04-22) and the `doRedis` package on each system. The Redis server ran on the *master* GNU/Linux machine, as did our master R session.

The example bootstrapping code is shown in in the listing below. We use the Redis "count" key

Listing˜2: Simple Bootstrapping Example

```
library('doRedis')
registerDoRedis('jobs')
redisDelete('count')

# Set up some data
data(iris)
x <- iris[which(iris[,5] != 'setosa'), c(1,5)]
trials <- 100000
chunkSize <- 100

# Start some local workers
startLocalWorkers(n=2, queue='jobs')
setChunkSize(chunkSize)

# Run the example
r <- foreach(icount(trials), .combine=cbind, .inorder=FALSE) %dopar% {
  redisIncrBy('count',chunkSize)
  ind <- sample(100, 100, replace=TRUE)
  estimate <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
  coefficients(estimate)
}

removeQueue('jobs')
```

and the `redisIncrBy` function to track the total number of jobs run so far, as described below. We set the number of bootstrap trials to a ridiculously large number in order to get a long-running example for the purposes of illustration.

We use a new function called `setChunkSize` in the above example to instruct the workers to pull `chunkSize` tasks at a time from their work queue. Setting this value can significantly improve performance, especially for short-running tasks. Setting the chunk size too large will adversely affect load balancing across the workers, however. The chunk size value may alternatively be set using the `.options.redis` options list directly in the `foreach` function as described in the documentation.

Once the above example is running, the workers update the total number of tasks taken in a Redis value called "count" at the start of each loop iteration. We can use another R process to visualize a moving average of computational rate. We ran the performance visualization R code in Listing 3 on the "master" workstation after starting the bootstrapping example (it requires the `xts` time-series package).

It is straightforward to add new workers to the work queue at any time. The following example R

Listing˜3: Performance Visualization

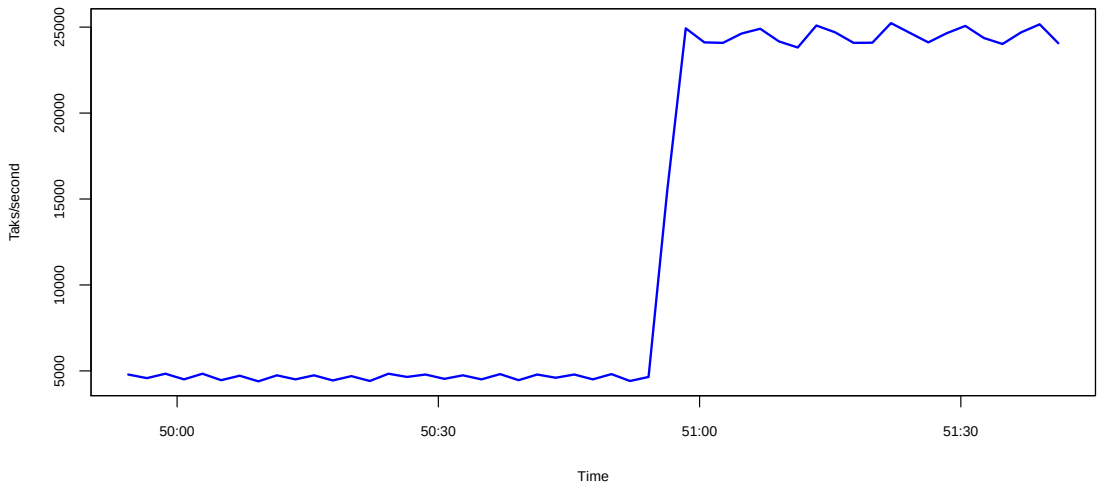
```
library('xts')
library('rredis')
redisConnect()
l <- 50
t1 <- Sys.time()
redisIncrBy('count',0)
x0 <- as.numeric(redisGet('count'))
r <- as.xts(0,order.by=t1)
while(TRUE)
 {
   Sys.sleep(2)
   x <- as.numeric(redisGet('count'))
   t2 <- Sys.time()
   d <- (x-x0)/(difftime(t2,t1,units="secs")[[1]])
   r <- rbind(r, as.xts(d, order.by=t2))
   t1 <- t2
   x0 <- x
   if(nrow(r)>l) r <- r[(nrow(r)-l):nrow(r),]
   plot(as.zoo(r),type='l',lwd=2,col=4, ylab='Tasks/second', xlab='Time')
 }
```

code illustrates adding four workers to the "jobs" work queue available on the host system "master":

Listing˜4: Adding Additional Workers

```
library('doRedis')
startLocalWorkers(n=4, queue='jobs', host='master')
```

We started the example bootstrap code running on the "master" system and the logged in to the much more powerful Windows Server 2003 system and added four additional workers using the above code. The performance plot clearly illustrates the dramatic increase in computational rate when the new workers were added:



3.3 A Parallel boot Function

Listing 5 presents a parallel capable variation of the `boot` function in the `boot` package. The `bootForEach` function uses `foreach` to distributed bootstrap processing to available workers. It has two more arguments than the standard `boot` function: `chunks` and `verbose`. Set `verbose=TRUE` to enabled back end worker process debugging. The bootstrap resampling replicates will be divided into `chunks` tasks for processing by `foreach`.

The example also illustrates the use of a custom combine function in the `foreach` loop.

4 A Few Technical Details

4.1 Random Number Generator Seeds

The initialization of pseudorandom number generators is an important consideration, especially when running simulations in parallel. Each `foreach` loop iteration (task) is assigned a number in order from the sequence $1, 2, \ldots$. By default, `doRedis` workers initialize the seed of their random number generator with a multiple of the first task number they receive. The multiple is chosen to very widely separate seed initialization values. This simple scheme is sufficient for many problems, and comparable to the initialization scheme used by many other parallel back ends.

The `doRedis` package includes a mechanism to define an arbitrary random seed initialization function. Such a function could be used, for example, with the `SPRNG` library.

The user-defined random seed initialization function must be called `set.seed.worker`, take one argument and must be exported to the workers explicitly in the `foreach` loop. The example shown in Listing 6 illustrates a simple user-defined seed function.

4.2 Redis Keys Used

The "job queue" name specified in the `registerDoRedis` and `redisWorker` functions is used as the root name for a family of Redis keys. The keys are defined by `<queue name>.*`–thus, every Redis key beginning with the queue name followed by period should be considered reserved. The keys have various uses, for example the `<queue>.count` key keeps an estimate of the number of active worker processes registered to take work from the queue.

Back end worker processes run a worker loop that blocks on work from one or more job queues. Periodically, the worker process checks for existence of a `<queue>.live` key. If the worker finds this key missing, it terminates the worker loop and deletes all Redis variables associated with the queue. A master R process may terminate workers and force the key cleanup using the `removeQueue` command.

4.3 Miscellaneous Details

If CTRL+C is pressed while a `foreach` loop is running, connection to the Redis server may be lost or enter an undefined state. An R session can reset connection to a Redis server at any time by issuing `redisClose()` followed by re-registering the `doRedis` back end.

Listing˜5: Parallel boot Function

```
`bootForEach` <- function (data, statistic, R, sim = "ordinary", stype = "i",
                 strata = rep(1, n), L = NULL, m = 0, weights = NULL,
                 ran.gen = function(d, p) d, mle = NULL, simple=FALSE,
                 chunks = 1, verbose=FALSE, ...)
{
  thisCall <- match.call()
  n <- if (length(dim(data)) == 2) nrow(data)
  else length(data)
  RB <- floor(R/chunks)

  combo <- function(...)
  {
    al <- list(...)
    out <- al[[1]]
    t <- lapply(al, "[[", "t")
    out$t <- do.call("rbind", t)
    out$R <- R
    out$call <- thisCall
    class(out) <- "boot"
    out
  }

  # We define an initial bootstrap replicate locally. We use this
  # to set up all the components of the bootstrap output object
  # that don't vary from run to run. This is more efficient for
  # large data sets than letting the workers retun this information.
  binit <- boot(data, statistic, 1, sim = sim, stype = stype,
               strata = strata, L = L, m = m, weights = weights,
               ran.gen = ran.gen, mle=mle, ...)

  foreach(j=icount(chunks), .inorder=FALSE, .combine=combo, .init=binit,
          .packages=c("boot","foreach"), .multicombine=TRUE, .verbose=verbose)
  %dopar%
  {
    if(j==chunks) RB <- RB + R %% chunks
    res <- boot(data, statistic, RB, sim = sim, stype = stype,
               strata = strata, L = L, m = m, weights = weights,
               ran.gen = ran.gen, mle=mle, ...)
    list(t=res$t)
  }
}
```

Listing˜6: User-defined RNG initialization

```
# First, use the default initialization:

> startLocalWorkers(n=5,queue='jobs')
> registerDoRedis('jobs')
> foreach(j=1:5,.combine='c') %dopar% runif(1)
 [1] 0.27572951 0.62581389 0.90845008 0.49669130 0.06106442

# Now, let's make all the workers use the same random seed initialization:

> set.seed.worker <- function(n) set.seed(55)
> foreach(j=1:5,.combine='c',.export='set.seed.worker') %dopar% runif(1)
[1] 0.5478135 0.5478135 0.5478135 0.5478135 0.5478135
```