Introduction to the `doRedis` Package

Bryan W. Lewis
blewis@illposed.net

August 6, 2015

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 independently of specific parallel computing 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 queue-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 key differentiating features of `doRedis` are elasticity and fault-tolerance, and portability across operating system platforms. The `doRedis` package is well-suited to small to medium-sized parallel computing jobs, especially across ad hoc collections of computing resources.

- `doRedis` allows for dynamic pools of workers. New workers may be added at any time, even in the middle of running computations. This feature is geared for 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.
- `doRedis` computations are partially fault tolerant. Failure of back-end worker R processes (for example due to a machine crash or simply scaling back elastic resources) are automatically detected and the affected tasks are automatically re-submitted.

- doRedis makes it particularly easy to run parallel jobs across different operating systems. It works equally well on GNU/Linux, Mac OS X, and Windows systems, and should work well on most POSIX 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.
- Like all foreach parallel back-ends, intermediate results may be aggregated incrementally, significantly reducing required memory overhead for problems that return large data.

2 Obtaining and Configuring the Redis server

Redis is an extremely popular open source networked key/value database, and operating system-specific packages are available for all major operating systems, including Windows.

For more information see: http://redis.io/download.

The Redis server is completely configured by the file redis.conf. It's important to make sure that the timeout setting is set to 0 in the redis.conf file when using doRedis. You may wish to peruse the rest of the configuration file and experiment with the other server settings as well.

3 doRedis Examples

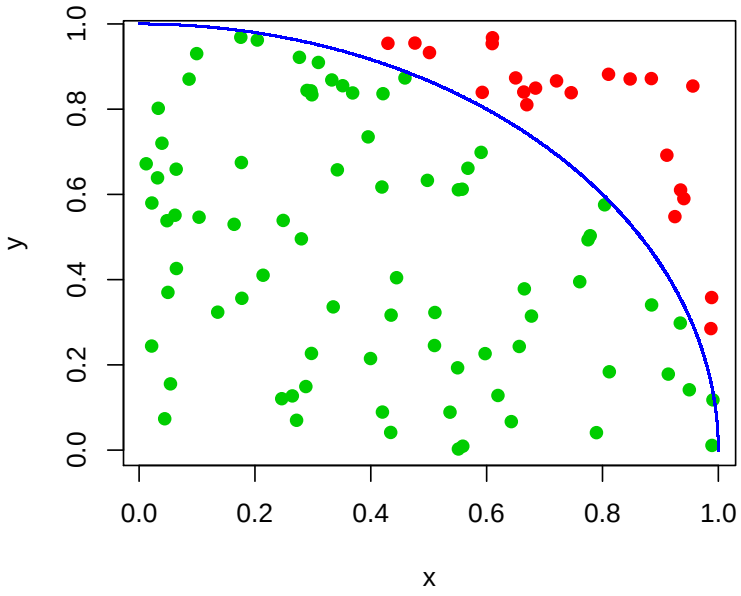
Let's start by exploring the operation of some doRedis features through a few examples. Unless otherwise noted, we assume that Redis is installed and running on the local computer ("localhost").

3.1 A Really 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 ("tasks" in `doRedis`) of the loop computes a scaled approximation of π 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. Each `doRedis` *job* is composed of a set of one or more *tasks*. Worker R processes listen on the work queue for new jobs. 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). R can issue work to a work queue even if there aren't any workers yet.

The next line:

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

starts up two worker R sessions on the local computer, both listening for work on the queue named "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 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, but the number returned should only be considered to be an estimate of the actual number of available workers.

The next lines actually run the 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 in this example) using the `icount` iterator function. 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 Fault tolerance

Parallel computations managed by `doRedis` tolerate failures among the back end worker R processes. Examples of failures include crashed back end R sessions, operating system crash or reboot, power outages, or simply dialing down elastic workers by turning them off. When a failure is detected, affected tasks are automatically re-submitted to the work queue. The option `ftinterval` controls how frequently `doRedis` checks for failure. The default value is 15 seconds, and the minimum allowed value is three seconds. (Very frequent checks for failure increase overhead and will slow computations down–the default value is usually reasonable.)

Listing 2 presents a contrived, but entirely self-contained example of fault tolerance. Verbose logging output is enabled to help document the inner workings of the example.

Listing 2: Fault Tolerance Example

```
> require("doRedis")
> registerDoRedis("jobs")
> startLocalWorkers(n=4,queue="jobs",timeout=1)
> cat("Workers started.\n")
> start <- Sys.time()
> x <- foreach(j=1:4, .combine=sum, .verbose=TRUE,
>              .options.redis=list(ftinterval=5, chunkSize=2)) %dopar%
>       {
>         if(difftime(Sys.time(),start) < 5) quit(save="no")
>         j
>       }

> removeQueue("jobs")
```

The example starts up four local worker processes and submits two tasks to the work queue "jobs." (There are four loop iterations, but the `chunkSize` option splits them into two tasks of two iterations each.) The parallel code block in the `foreach` loop instructs worker processes to quit if less than 5 seconds have elapsed since the start of the program. Note that the `start` variable is defined by the master process and automatically exported to the worker process R environment by `foreach`–a really nice feature! The termination criterion will affect the first two workers that get tasks, resulting in their immediate exit and simulating crashed R sessions.

Meanwhile, the master process has a fault check period set to 5 seconds (the `ftinterval=5` parameter), and after that interval will detect the fault and re-submit the failed tasks.

The remaining two worker processes pick up the re-submitted tasks, and since the time interval will be sufficiently past the start, they will finish the tasks and return their results.

The fault detection method is simple but robust, and described in detail in Section 4.

3.3 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 the following examples.

Listing 3: 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")
```

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 and the `doRedis` package on each system (this example was run in April, 2010! but it's still relevant). The Redis server ran on the *master* GNU/Linux machine, as did our master R session. The example bootstrapping code is shown in Listing 3.

We use the Redis "count" key 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 very 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 package 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 4 on the "master" workstation after starting the bootstrapping example (it requires the `xts` time-series package).

Listing 4: 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")
> }
```

Listing 5: Adding Additional Workers

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

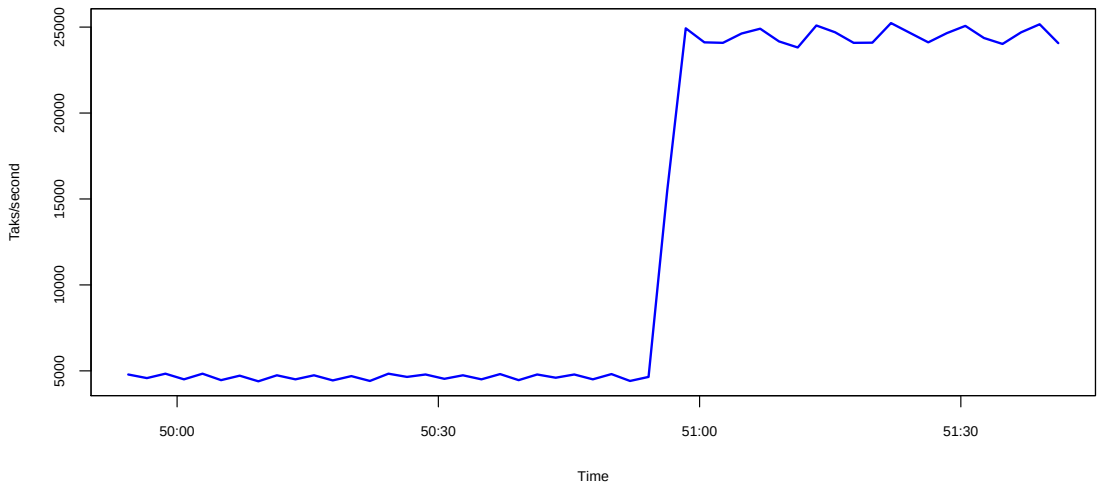


Figure 1: Performance gain after adding workers to a running computation

We started the example bootstrap code running on the "master" system and the logged in to the much more powerful Windows Server system and added four additional workers using code in Listing 5. The resulting performance

plot clearly illustrates the dramatic increase in computational rate when the new workers were added.

3.4 A Parallel boot Function

Listing 6 presents a parallel capable variation of the `boot` function from 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.

Listing 6: 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)
>   if(R<2) stop("R must be greater than 1")
>   Rm1 <- R - 1
>   RB <- floor(Rm1/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 return 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 + Rm1 %% 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)
>  }
> }
```

4 Technical Details

A `foreach` loop iteration is a parameterized R expression that represents a unit of work. The expression is the body of the `foreach` loop and the parameter, if it exists, is the loop variable value. (It is possible to use `foreach` non-parametrically with an anonymous iterator to simply replicate the loop body expression a number of times.) Each iteration is enumerated so that `foreach` can put the results together in order if required.

A *task* is a collection of one or more loop iterations. The number of iterations per task is at most `chunkSize`.

A *job* is a collection of one or more tasks that covers all the iterations in the `foreach` loop. Each job is assigned a unique identifier.

Jobs are submitted as a sequence of tasks to a *work queue*–technically, a special Redis key called a "list" that supports blocking reads. Users choose the name of the work queue in the `registerDoRedis` function. Master R programs that issue jobs wait for results on yet another blocking Redis list, a result queue associated with the job.

R workers listen on work queues for tasks using blocking reads. As shown in the last section the number of workers is dynamic. It's possible for workers to listen on queues before any jobs exist, or for masters to issue jobs to queues without any workers available.

4.1 Redis Key Organization

The "work queue" name specified in the `registerDoRedis` and `redisWorker` functions is used as the root name for a family of Redis keys used to organize computation. Figure 2 illustrates example Redis keys used by a master and worker R processes for a work queue named "myjobs", described in detail below.

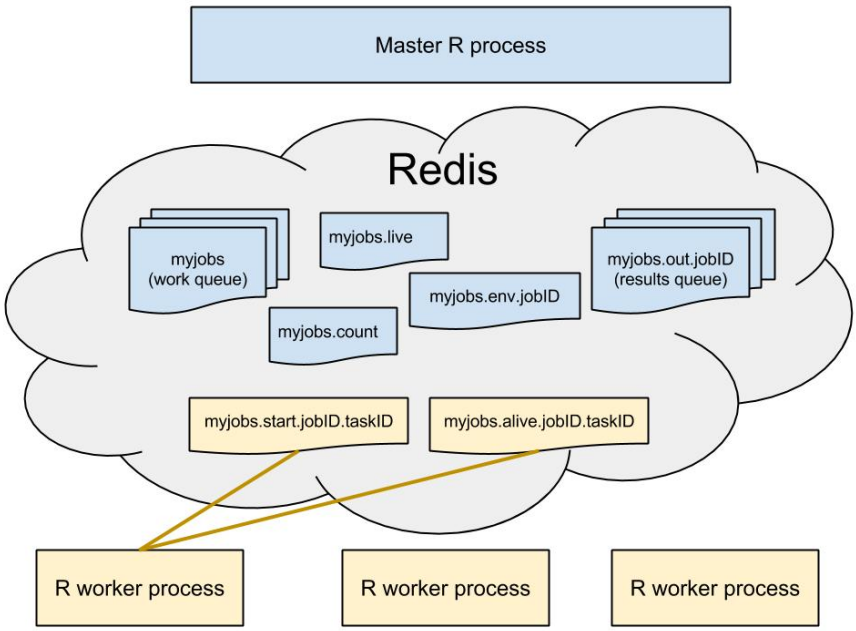


Figure 2: Example `doRedis` keys for an example work queue named "myjobs"

The name of the work queue illustrated in Figure 2 is "myjobs." The corresponding Redis key is also named

"myjobs" and it's a Redis list value type (that is, a queue). Such a queue can be set up, for example, with `registerDoRedis(queue="myjobs")`.

Removal of the "jobs.live" key serves as a signal that the work queue has been removed, for example with the `removeQueue("myjobs")` function. After this happens, R workers listening on the queue will clean up any Redis keys that they created and terminate after a timeout period.

A counter key named "myjobs.count" enumerates the number of R worker processes registered on the queue. It is only an estimate of the number of workers currently registered to accept work on the queue.

`foreach` assigns every job an R environment with state required to execute the loop expression. The example in Figure 2 illustrates a job environment for the "myjobs" queue and the "jobID" job called "myjobs.env.jobID" R worker processes working on "jobID" will download this environment key once (independently of the number of tasks they run for the job).

`doRedis` pushes the tasks (loop iterations) associated with "jobID" into the "myjobs" queue using the labels "jobID.n", where n is the number of each task, n=1, 2, ...

R workers listen on the "myjobs" queue for tasks. It's a blocking queue, but the workers periodically time out. After each time out they check to see if the "myjobs.live" key still exists, and if it doesn't they clean up their Redis keys and terminate. If it's still there, they loop and listen again for jobs.

When a job/task announcement arrives in the "myjobs" queue, a worker downloads the new task from the "myjobs" queue. The worker then:

1. Checks the job ID to see if it already has the job environment. If it doesn't it downloads it (from "myjobs.env.jobID" in the example) and initializes a new R environment specific to this job ID.
2. The R worker process initializes a task started key, illustrated in Figure 2 as "myjobs.start.jobID.taskID".
3. The R worker process initializes a thread that maintains a task liveness key, illustrated in Figure 2 as "myjobs.alive.jobID.taskID".
4. When a task completes, the worker places the result in a job result queue for that job ID, shown in Figure 2 as "myjobs.out.jobID", and then removes its corresponding start and alive keys.

Meanwhile, the R master process is listening for results on the job result queue, shown in Figure 2 as "myjobs.out.jobID" Results arrive from the workers as R lists of the form `list(id=result)`, where `id` corresponds to the task ID number and `result` to the computed task result. `foreach` can use the task ID number to cache and order results, unless the option `.inorder=FALSE` was specified.

4.2 Worker Fault Detection and Recovery

While running, each task is associated with two keys described in the last section: a task "start" key and a task "alive" key. The "start" key is created by an R worker process when a task is downloaded. The "alive" key is a Redis ephemeral key with a relatively short time out (after which it disappears). Each R worker processes maintains a background thread that keeps the ephemeral "alive" key active while the task runs. If for some reason the R worker process crashes, or the work system crashes or reboots, or the network fails, then the "alive" key will time out and be removed from Redis.

After `foreach` sets up a job, the master R process listens for results on the associated job ID results queue. It's a blocking read, but the master R process periodically times out. After each time out, the master examines all the "start" and "alive" keys associated with the job. If it finds an imbalance in the keys (a start key without a

corresponding alive key), then the master R process assumes that task has failed and resubmits the failed task to the work queue.

It's possible that a wayward R worker process might return after its task has been declared lost. This might occur, for example, under intermittent network failure. In such cases, results for tasks might be returned more than once in the result queue, but `doRedis` is prepared for this and simply discards repeated results for the same task ID.

4.3 Random Number Generator Seeds

The initialization of pseudorandom number generators is an important consideration, especially when running simulations in parallel. By default, workers use the L'Ecuyer-CMRG method from R's `parallel` package. Each `foreach` loop iteration is assigned a L'Ecuyer stream making parallel random number generation reproducible regardless of the number of workers or chunk size settings. Simply set the random seed prior to a `foreach` loop as you would do in a sequential program.

Note that, although `doRedis` uses the L'Ecuyer random number generator internally, it records and restores the state and kind of random number generator prior to the `foreach` loop.

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 or with the `doRNG` package for `foreach`.

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 7 illustrates a simple user-defined seed function.

Listing 7: User-defined RNG initialization

```
> startLocalWorkers(n=5,queue="jobs")
> registerDoRedis("jobs")

> # Make all the workers use the same random seed initialization and the old
> # "Super-Duper" RNG:
> set.seed.worker <- function(n) {
+   RNGkind("Super-Duper")
+   set.seed(55)
+ }
> foreach(j=1:5,.combine="c",.export="set.seed.worker") %dopar% runif(1)
[1] 0.2115148 0.2115148 0.2115148 0.2115148 0.2115148
```

4.4 Known Problems and Limitations

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.

Redis limits database values to less than 512 MB. Neither the `foreach` loop parameters nor the job environment may exceed this size. If you need to work on chunks of data larger than this, see the Advanced Topics section for examples of working on already distributed data in place.

5 Advanced Topics

Let's start this section by dealing with the 512 MB Redis value size limit. Problems will come along in which you'll need to get more than 512 MB of data to the R worker processes. There are several approaches that one might take:

1. Distribute the data outside of Redis, for example through a shared distributed file system like PVFS or Lustre, or through another database.
2. Break the data up into chunks that each fit into Redis and use Redis to distribute the data.

The first approach is often a good one, but is outside of the scope of this vignette. We illustrate the second approach in the following example. For the purposes of illustration, the example matrix is tiny and we break it up into only two chunks. But the idea extends directly to very large problems.

Listing 8: Explicitly breaking a problem up into chunks

```
> registerDoRedis("jobs")

> set.seed(1)
> A <- matrix(rnorm(100),nrow=10)   # (Imagine that A is really big.)

# Partition the matrix into parts small enough to fit into Redis values
# (less than 2GB). We partition our example matrix into two parts:

> A1 <- A[1:5,]      # First five rows
> A2 <- A[6:10,]     # Last five rows

# Let's explicitly assign these sub-matrices as Redis values. Manually breaking
# up the data like this helps avoid putting too much data in the R environment
# exported by foreach.

> redisSet("A1", A1)
> redisSet("A2", A2)

> ans <- foreach(j=1:2, .combine=c) %dopar% {
>   chunk <- sprintf("A%d",j)
>   mychunk <- redisGet(chunk)
>   sum(mychunk)
> }

> print(ans)
[1] 6.216482 4.672254
```

The point of the example in Listing 8 is that the workers explicitly download just their portion of the data inside the `foreach` loop. This avoids putting the data into the exported R environment, which could exceed the Redis 2 GB value size limit. The example also avoids sending data to workers that they don't need. Each worker downloads just the data it needs and nothing more.

5.1 worker.init

If you export a function named `worker.init` that takes no arguments, it will be run by the workers once just after downloading a new job environment. The function may contain any worker initialization code not be appropriate for the body of the `foreach` loop that only needs to run once per job, independently of the number of tasks that might run on a particular worker.