Introduction to the `doRedis` Package

Bryan W. Lewis
blewis@illposed.net

August 5, 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

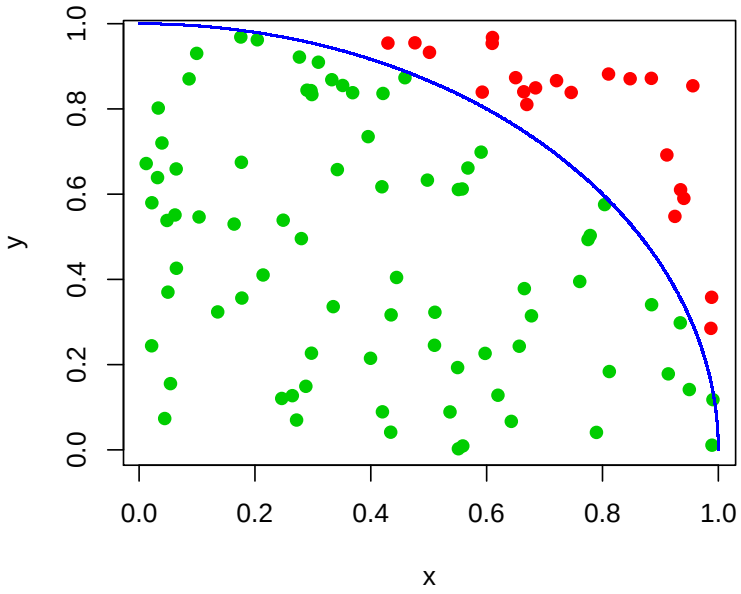
Let's start by exploring the operation of some `doRedis` features through a few examples. Unless otherwise noted, we assume that Redis is installed and running on the local computer ("localhost").

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 30 seconds, and the minimum allowed value is one second. (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.

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*.

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")
```

- 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`

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")
```

time-series package).

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.

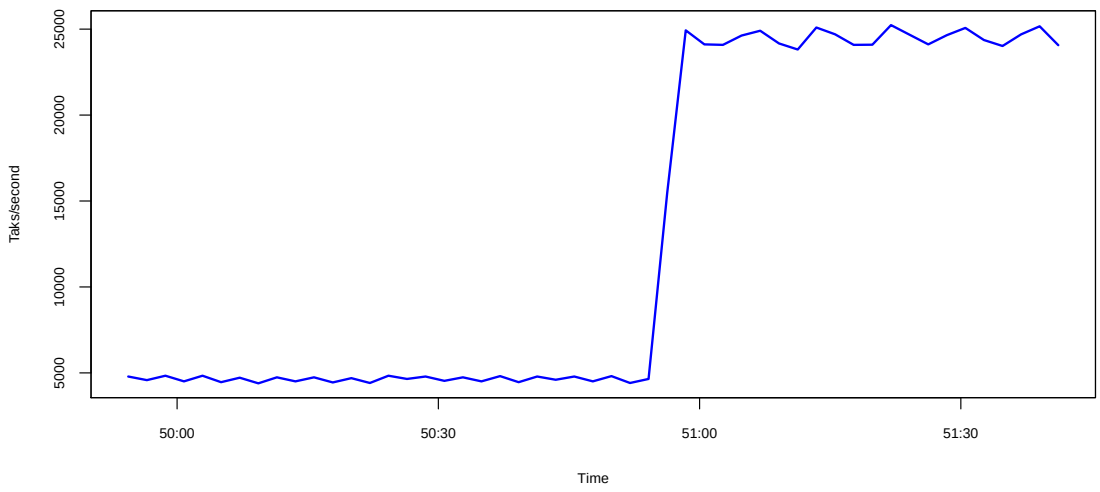


Figure 1: Performance gain after adding workers to a running computation

4 Technical Details

A `doRedis` *task* is a parameterized R expression that represents a unit of work. The task expression is the body of a `foreach` loop. A single task may consist of one or more loop iterations depending on the value of the `chunkSize` parameter (the default is one).

All of the tasks associated with a single `foreach` statement are collected into a `doRedis` *job*. Each job is assigned a numeric identifier, and each job's tasks are numbered $1, 2, 3, ...$

Jobs are announced on 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 specific to the job.

R workers listen on work queues for jobs 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, described in detail below.

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)
>  }
> }
```

The name of the work queue illustrated in Figure 2 is "jobs." The corresponding Redis key is also named "jobs" and it's a Redis list value type (that is, a queue). Such a queue can be set up,

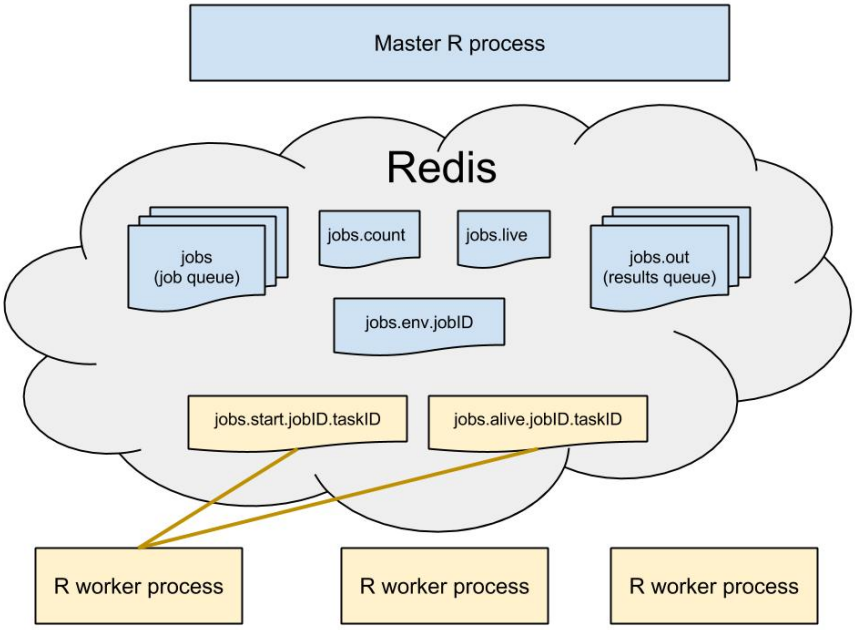


Figure 2: Example doRedis keys

for example, with `registerDoRedis(queue="jobs")`.

Along with the work queue, a counter key named "jobs:counter" is set up to enumerate jobs. The key is atomically incremented by Redis with each new job. Removal of "jobs:counter" key serves as a signal that the work queue has been removed, for example with the `removeQueue("jobs")` function. After this happens, any workers listening on the queue will terminate.

`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 "jobs" queue and job number 1 called "jobs:1.env." R worker processes working on job number 1 in the "jobs" queue will download this environment key once (independently of the number of tasks they run for the job).

All of the tasks for the job number 1 in the "jobs" queue are placed in a Redis key called "jobs:1" in Figure 2. The Redis value type associated with this key is a hash table. Hash table entries are named by task number $(1, 2, ...)$ and the hash table values contain the `foreach` loop parameters associated with the task.

The `foreach` loop performs the following steps to create a new job in the "jobs" queue consisting of n tasks:

1. Increment "jobs:count", recording the new job number (say, job number 1 for example).

2. Create and upload the "jobs:1.env" job environment.

3. Populate the n tasks in the "jobs:1" hash table with the `foreach` loop parameters for each task.

4. Push n copies of the job number to the "jobs" queue to announce the new job with n tasks.

R workers listen on the "jobs" queue for new jobs. It's a blocking queue, but the workers periodically time out. After each time out they check to see if the "jobs:counter" key still exists, and if it doesn't they terminate. If it's still there, they loop and listen again for jobs.

When a job announcement arrives, a worker downloads the new job number from the "jobs" queue. The worker then:

1. Checks the job number to see if it already has the job environment. If it doesn't it downloads it (from "jobs:1.env" in the example).

2. Calls the `getTask` function defined in the job environment to get a task. By default, this function pulls tasks in order from the associated job task hash table ("jobs:1" in Figure 2). The function can be customized to do more (see the advanced examples topic in the sequel).

3. Upon successfully downloading a task, Redis creates a task start key that indicates that the task is being processed. The example in Figure 2 illustrates a task key called "jobs:1.start.5" indicating task number 5 of job number 1 in the jobs queue.

4. The R worker process maintains a task liveness key, illustrated in Figure 2 as "jobs:1.alive.5".

5. When a task completes, the worker places the result in the job result queue, shown in Figure 2 as "jobs:1.results", 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 "jobs.1.results." 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.

4.2 Worker Fault Detection and Recovery

While running, each task is associated with two keys described in the last section: a task "started" key and a task "alive" key. The "started" key is normally created by Redis when a task is downloaded. The "alive" key is a Redis ephemeral key with a relatively short time out (after which it disappears). The R worker processes maintain background threads that keep the ephemeral "alive" key active while the job is being run. If for some reason the R worker process crashes, or the work system crashes or reboots, or 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 results queue. It's a blocking read, but the master R process periodically times out. After each time out, the master examines all the "started" and "alive" keys associated with the job. If it finds a mismatch, the master R process assumes that task has failed and:

1. resubmits the task in the corresponding job tasks hash table;

2. announces another task for this job in the job queue.

It's possible that a wayward R worker process might return after its task has been declared lost. 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.

4.3 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.

Listing 7: 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
```

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.

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 2 GB. 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 2 GB Redis value size limit. Problems will come along in which you'll need to get more than 2 GB 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.

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.

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 Custom task assignment

The example shown in Listing 8 presents one way to distribute just the data needed to each task. But what if we need to repeat the `foreach` loop in Listing 8, for example in the running of an iterative method? It would be nice if the workers could cache their chunks of data locally and simply re-use that without fetching new data.

The default `doRedis` approach assigns work to workers on a first-come, first-served basis. But it also allows for custom task assignment, for example to align tasks with existing cached data on workers. Custom task assignment can also be used to allocate tasks based on distance to already distributed data.

Let's recall how tasks are assigned to a worker R process by `doRedis` (refer to Figure 2):

1. The worker waits for a job announcement with a blocking read on the job work queue ("jobs" in Figure 2).

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
```

2. After a job ID is removed from the work queue, the worker downloads the job R environment value ("jobs:1.env").

3. The environment contains a function called `getTask`. The worker invokes this function with the work queue name, the job ID, and a worker "tag" value to get a task.

4. If a task was obtained, the worker executes it, placing the result in the result queue ("jobs.1.results"). Otherwise the worker replaces the obtained job ID into the work queue and repeats (see below for a discussion of this exceptional case).

Customization of task assignment boils down to supplying a custom `getTask` and setting a "tag" value in the workers.

5.1.1 Setting worker tags

A "tag" is a character string that workers report to their `getTask` functions. Tags can be any character string, and they do not need to be unique. Workers report their system host names by

default. Interesting tag possibilities are chunk number and worker I.P. address.

Set tags with the `setTag` function. This step is usually done once in the first `foreach` loop. Once a tag is set, it remains in force until reset (the tag value will not vary from job to job).

5.1.2 Customizing the getTask function

The task assignment decision is made by the Redis server. The default `getTask` function specifies a Lua script that is run by Redis. It ignores the "tag" argument supplied by the workers. Here it is: Note that it's important for the `getTask` function to set the task started key (the Lua statement

Listing 9: Default getTask function

```
getTask <- function(queue, job_id, ...)
{
  key <- sprintf("
  redisEval("local x=redis.call('hkeys',KEYS[1])[1];
             if x==nil then return nil end;
             local ans=redis.call('hget',KEYS[1],x);
             redis.call('set', KEYS[1] .. '.start.' .. x, x);
             redis.call('hdel',KEYS[1],x);i
             return ans",key)
}
```

`redis.call('set', KEYS[1] .. '.start.' .. x, x)`). For reliable operation, this should be set by Redis and not left up to the R worker.

Let's modify the example in Listing 8 to preferentially assign work to workers that already have locally cached data. With these modifications in place, the `foreach` loop in Listing 8 can be repeated without incurring any new data movement costs as long as the `doRedis` worker pool remains constant.

The custom `myGetTask` function shown in Listing 10 first tries to assign a specific task number requested by the worker via its tag. If that fails, it assigns the first available task or NULL if there aren't any tasks. Listing 11 shows a modified version of the `foreach` loop from example 8. The modifications add the following:

- Cache the data in the worker's R global environment the first time run.
- Assign a tag corresponding to the cached data chunk.
- Use the custom `myGetTask` function from Listing 10.
- Add some debugging output to help understand what's happening.

Let's say that the example shown in Listing 11 is run using the "jobs" work queue with two local `doRedis` workers, for example with: The first time you run the example `foreach` loop in Listing

Listing 10: Custom getTask function

```
> myGetTask <- function(queue, job_id, tag, ...)
{
# The default task Redis hash map is labeled queue:job_id:
>   key <- sprintf("%s:%s",queue, job_id)
# The worker tag is passed in to the Redis Lua interpreter as ARGV[1].
# See ?redisEval for details.
>   redisEval("local x=redis.call('hkeys',KEYS[1])[tonumber(ARGV[1])];
+             if x==nil then x=redis.call('hkeys',KEYS[1])[1]; end;
+             if x==nil then return nil; end;
+             local ans=redis.call('hget',KEYS[1],x);
+             redis.call('set', KEYS[1] .. '.start.' .. x, x);
+             redis.call('hdel',KEYS[1],x);
+             return ans",key,FALSE,tag)
}
```

Listing 11: Modified foreach loop

```
> ans  <-
> foreach(j=1:2, .options.redis=list(getTask=myGetTask)) %dopar% {
>   chunk <- sprintf("A%d",j)
>   log <- paste("My job is to sum chunk ",chunk)
>   log <- paste(log,"My tag ID is ",doRedis:::.getTag())
# Check if we have a cached copy of the chunk
>   if(exists(chunk))
>   {
>     log <- paste(log,"I've already got this chunk...")
>     mychunk <- get(chunk)
>     return(list(sum=sum(mychunk), log=log))
>   }
# We don't have it cached, download it and cache it locally...
>   log <- paste(log,"Downloading chunk",chunk)
>   setTag(j)     # Set our tag!
>   assign(chunk, redisGet(chunk), envir=globalenv())
>   mychunk <- get(chunk, envir=globalenv())
>   list(sum=sum(mychunk), log=log)
> }
> print(ans)
```

Listing 12: Starting two local R workers

```
> library("doRedis")
> registerDoRedis("jobs")
> startLocalWorkers(n=2, queue="jobs")
```

Listing 13: First foreach loop run

```
[[1]]
[[1]]$sum
[1] 0

[[1]]$log
[1] "My job is to sum chunk  A1 My tag ID is localhost Downloading chunk A1"


[[2]]
[[2]]$sum
[1] 0

[[2]]$log
[1] "My job is to sum chunk  A2 My tag ID is localhost Downloading chunk A2"
```

11, you'll see output similar to that shown in Listing 13 below. If you repeat the `foreach` loop in Listing 11, you'll see output similar to that shown below in Listing 14, which shows that the workers are using their cached data. You can repeat the `foreach` loop as many times as you like and the output should remain constant.

Listing 14: Repeated foreach loop runs

```
[[1]]
[[1]]$sum
[1] 0

[[1]]$log
[1] "My job is to sum chunk  A1 My tag ID is  1 I've already got this chunk..."


[[2]]
[[2]]$sum
[1] 0

[[2]]$log
[1] "My job is to sum chunk  A2 My tag ID is  2 I've already got this chunk..."
```

Note that, even if one of the workers quits or crashes, the remaining worker will be assigned work to finish the job. And, the method still accommodates additional workers coming on line at any time.

5.2 Things to watch out for

`foreach` submits all tasks associated with one job in a single Redis transaction. This is important in light of the custom task pulling example in Listing 10. We'd like to make sure that all the tasks are available before doling out work to avoid prematurely assigning a task to the wrong worker.

However, that means that if the `foreach` loop parameterization contains a lot of data, there can be a significant lag before starting the work. Avoid loop parameterizations containing lots of data. It's a better idea to manually break up data into Redis keys as shown in the example in Listing 8.

It's possible that a worker can remove a job announcement from a work queue without requesting a task via `getTask` from the job's task set. That's a problem. Another problem already discussed in the fault tolerance section occurs when the number of started tasks does not match the number of alive tasks. The master process periodically checks two things:

1. Number of started tasks = number of alive tasks (fault check)
2. queued + started + finished = total

If item 2 fails, then the queue is out of balance and the master R program resubmits the missing number of task entries. In either case, it's possible that a rogue worker process tricks us and completes work that apparently failed. The master R program running the `foreach` loop throws away replicated task results, keeping only the first result to arrive.

5.3 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. This is a generic function that may contain any worker initialization code that may not be appropriate for the body of the `foreach` loop. The `worker.init` function is rarely needed.