Journal of Statistical Software

March 2011, Volume 39, Code Snippet 1. *http://www.jstatsoft.org/*

Passing in Command Line Arguments and Parallel Cluster/Multicore Batching in R with batch

Thomas J. Hoffmann
University of California, San Francisco

Abstract

It is often useful to rerun a command line R script with some slight change in the parameters used to run it – a new set of parameters for a simulation, a different dataset to process, etc. The R package **batch** provides a means to pass in multiple command line options, including vectors of values in the usual R format, easily into R. The same script can be setup to run things in parallel via different command line arguments. The R package **batch** also provides a means to simplify this parallel batching by allowing one to use R and an R-like syntax for arguments to spread a script across a cluster or local multicore/multiprocessor computer, with automated syntax for several popular cluster types. Finally it provides a means to aggregate the results together of multiple processes run on a cluster.

Keywords: parallel, cluster, command line arguments, batch, R.

1. Introduction

With multicore CPUs (central processing units) prevalent in even the most budget computers, clusters becoming more cost effective and realistic in smaller scenarios, and even cloud computing becoming more common, it is useful to have multiple different ways to parallelize ones code. There are many different methods of implementing parallelism into your R code (R Development Core Team 2010) to make it run faster. Depending on how the code is written, different methods to make it run in parallel will be easier to use. For a good overview of the current state of parallel programming in R see Schmidberger *et al.* (2009), and the Comprehensive R Archive Network (CRAN) task view for high performance computing (Eddelbuettel 2010).

One common method is to replace the `lapply` function with something that runs it in parallel. The R package **multicore** (Urbanek 2011) accomplishes this on a non-Microsoft Windows local

machine with the command `mclapply`. With `mclapply` a function is parallelized across a list, utilizing any data already in the workspace; however the pieces cannot interact with each other or cause any modification to any existing values other than what is returned, as is necessary for parallel methods. A convenience method is provided in the **foreach** package for general iteration over elements (REvolution Computing 2009). The packages **snowfall** and **sfCluster** (Knaus *et al.* 2009), built on the package **snow** (Rossini *et al.* 2007), extends this functionality to a cluster (it can also be run locally). Again, a function is parallelized across a list. However, **snowfall** does not use the current workspace, and instead any data must be first exported before being used by the parallel list function. The package abstracts the usage of the cluster.

On the other hand, there are lower-level routines to access parallel functionality, i.e., **fork** (Warnes 2009), but these are typically not as user-friendly. Other options include **Rmpi** (Yu 2002) to access the standard MPI framework for parallel computing (Burns *et al.* 1994; Squyres and Lumsdaine 2003; Gabriel *et al.* 2004; Gropp *et al.* 1996). There are also specific implementations for certain clusters, i.e., **Rlsf** for LSF computing (Smith *et al.* 2007).

The R package **batch** lies inbetween that of the lower and higher level cluster packages. First the package **batch** provides a means to automate parsing command line arguments in R, allowing users to easily override any default values already specified in an R script. Parameters can be passed in as numerical values, strings, or even vectors of values. Then by using these command line arguments, an alternative and intuitive method of implementing parallelism into your R code is to simply run the same R script multiple times. Each time the script is run, it can be run with different command line arguments. For example, these could be different parameter values for a simulation. Or for a simulation that takes too long to run, they could be multiple runs of the same simulation, just with different random number seeds passed in through the command line. It can be easier to simply run your script multiple times on a cluster, rather than using other cluster implementations. The package can also be run locally on any operating system, by default utilizing as many cores and CPUs as are in a system. It can be a particularly useful parallel implementation if some parameter values accidentally make your code crash; the rest of the parameter values will still run, as they are in separate processes. It also means that the instances of your program do not all need to share the same memory, and can load in pieces of a dataset at a time. Lastly, the R package **batch** provides a means for intuitively batching off multiple instances of your code with a convenient R syntax, automatically configuring some of the options for several common clusters. It should run on virtually any cluster that has some method for batching jobs. With no modification to your code, the same script can be used on a local multicore system when a cluster is not available or is occupied.

Compared to other parallel processing packages, the **batch** package has several advantages and disadvantages. Rather than running scripts in parallel by wrapping them in a function to be applied across a list (as in **multicore** or **snowfall**), one runs the script directly with different command-line values. This can be more intuitive, for example, in situations dealing with large datasets where the dataset does not fit in memory, and things must be explicitly chopped into pieces anyway. The package **batch** is also tightly integrated with the job scheduler, by adding convenience functions that submit jobs to the cluster. The options of each job, e.g., what priority queue, can be modified accordingly and dynamically after the jobs have been submitted. It is also useful and straightforward if you wish to further break your simulations apart to run pieces on multiple different clusters or computers you have access to. Intermediate

output of jobs that have been completed can be viewed, along with status updates printed by currently running jobs and the job scheduler. The R package **batch** provides a means to pass parameter values into scripts and run them in parallel on a cluster or locally on any operating system. The package is available from CRAN at http://CRAN.R-project.org/package=batch.

2. Passing in command line arguments

2.1. Simple example

We first begin with a simple example of how to override the value of the variables `a` and `b` from the command line in an R script file. Suppose we have the following simple code in the file '`test.R`':

```
a <- 10
b <- 20
library("batch")
parseCommandArgs()
print(a)
print(b)
```

We can run this script from the command prompt in several different ways, unix users may be most familiar with `R CMD BATCH test.R`. For Microsoft Windows users, you can run this from the command prompt, but the R `bin` directory must be in your path, or you must provide the entire path to the `R.exe` executable instead of `R` (i.e., '`C:\Program Files\R\R-`*version*`\bin\R.exe`' with the quotation marks). Alternatively to `R CMD BATCH`, we can run it with the following so the output goes directly to our screen, rather than to '`test.Rout`'.

```
R --vanilla < test.R
```

then we would get the following output:

```
> a <- 10
> b <- 20
> library("batch")
> parseCommandArgs()
> print(a)
[1] 10
> print(b)
[1] 20
```

Running other code with the `--vanilla` option will have further advantages in other code we will write later on. In particular, the `--vanilla` option prevents the output ('`.Rdata`' file) from being read before or saved after execution. When running in parallel this is typically not desirable, and will not function correctly. Instead we will want to save the output explicitly to a numerically incremented filename.

Now, in order to pass in a different numeric value for `a`, we can run the following

```
R --vanilla --args a 7 b 7 < test.R
```

Then we would get the following output:

```
> a <- 10
> b <- 20
> library("batch")
> parseCommandArgs()
$a
[1] 7

$b
[1] 7

> print(a)
[1] 7
> print(b)
[1] 7
```

The code `parseCommandArgs()` handles parsing the arguments passed in from the command line. The values are then assigned to the current environment, effectively overriding the default values of `a` and `b`. The return from the call to `parseCommandArgs()` function is a list of all of the parameters set in the command prompt, this gives quick feedback that the parameters really were set. Finally, if we want to get the values of the command line arguments, without actually overwriting the values in the current environment, we could change the fourth line to read

```
res <- parseCommandArgs(evaluate = FALSE)
```

Then only the list is returned, the value is not set. As a note, it is best to use variable names that aren't the names of R functions. For example, do not try to set the value of `beta` in your code, as it is also the name of an R function. Instead try `BETA`, for example.

We have a few more examples of passing in different values for `a`. In the following we pass a simple string, a more complicated string with a space in it, a vector of numeric values, and a vector of strings.

```
R --vanilla --args a foo < test.R
R --vanilla --args a "foo bar" < test.R
R --vanilla --args a "c(1, 10)" < test.R
R --vanilla --args a "c('foo', 'bar')" < test.R
```

Passing a vector is similar to the R syntax of creating a vector in R.

3. Running your code in parallel – A simulation case study

We will start off by writing a simple code script for a simulation as an example. Then, to break it in parallel, we run multiple instances of R with different command line options. We

provide a means to make the batching more user friendly as well. Here we go through a case study of how to run a simulation in parallel with the R package **batch**. Suppose we want to write a simple simulation that empirically wants to compute the power to detect a difference between two normal populations.

3.1. Our main simulation script file 'sim.R'

Although we split up the following code into several sections, it should all go sequentially into the file 'sim.R'. This script will run our simulation.

As before, we start off by giving default values to several parameters, and then running `parseCommandArgs()` to override those values.

```
seed <- 1000
n <- 50
nsim <- 10000
mu <- c(0, 0.5)
sd <- c(1, 1)
library("batch")
parseCommandArgs()
```

In this code, `seed` is the seed that we will set, `n` is the number of people in each population, `nsim` is the number of simulations to run, `mu` is the vector representing the mean of each of the populations, and `sd` is the vector of the standard deviation of each population. Note that we used `mu` rather than `mean <- c(0, 0.5)`, as `mean` is a function in R, and attempting to set the value for `mean` from the command line would cause an error.

After `parseCommandArgs()` has been run, we can set the value for the seed with

```
set.seed(seed)
```

Next in the file goes the simulation code. Here we are comparing two normal populations. So we draw up some normal populations for the variables X and Y and run a t-test on them with the following code:

```
pvalue <- rep(0, nsim)
for(i in 1:nsim) {
  X <- rnorm(n = n, mean = mu[1], sd = sd[1])
  Y <- rnorm(n = n, mean = mu[2], sd = sd[2])
  pvalue[i] <- t.test(X, Y)$p.value
}
power <- mean(pvalue <= 0.05)
```

At the end of this code we compute the empirical power of our simulations.

Lastly, we might want to store our results in a file. If we store the data as comma separated value (CSV) file, we will have the advantage of using other code to paste them together later. A CSV file is easily opened in any spreadsheet program. The following code saves our results to a CSV file:

```
out <- data.frame(seed = seed, nsim = nsim, n = n,
  mu = paste(mu, collapse = ","),
  sd = paste(sd, collapse = ","), power = power)
outfilename <- paste("res", seed, ".csv", sep = "")
print(out)
write.csv(out, outfilename, row.names = FALSE)
```

This completes our simulation script file.

3.2. Multiple parameters

Suppose we have a range of values for `mu` and `n` that we want to vary in '`sim.R`'. We could manually batch off command line arguments, or we could use the function `rbatch`. We could put the following code into '`param-sim.R`' to loop over several different effect sizes:

```
library("batch")
seed <- 1000
for(i in 1:10)
  seed <- rbatch("sim.R", seed = seed, n = 25, mu = c(0, i / 10))
rbatch.local.run()
```

The arguments to the function `rbatch` is the name of the R file to run, the seed to set, and then any other command line arguments you want to pass to the R file you are running. For example here `n = 25` sets the value of `n` to be `25` when '`sim.R`' is run. Through the loop, `mu` gets set to `0.1`, `0.2`, ..., `1`. When the code is to be run locally, you will need to follow the last `rbatch` function call with the code `rbatch.local.run()`. When this command is run locally, it will effectively act as a local job scheduler for each process, using as many cores and CPUs as available. If you leave this code in, but you are submitting on a cluster, then it will have no effect. On a cluster, the `rbatch` function directly submits a job to the cluster, which handles the job scheduling. Practically, you can always leave the command `rbatch.local.run()` in, and it will run only when submission is done locally.

To run this locally on a a Microsoft Windows machine, utilizing all cores and CPUs on the system (the default configuration), one can source or type in the commands from '`param-sim.R`' into the graphical interface for R (also true on MacOS X graphical interface to R). If the number of cores is incorrectly detected, then instead run `rbatch.local.run(4)`, e.g., for a quad core system.

To run this code in unix, again utilizing all cores, enter the following from the command prompt

```
R --vanilla < param-sim.R
```

However, you may also want to run this on a cluster. Most clusters will generally have a queuing system that you can submit jobs to. Behind the scenes, we are assembling the job submission string to pass to the queuing system. If you have LSF (Platform Computing Corporation 2001) or MOSIX (Barak and La'adan 1997) cluster configured, then things are simple. To run it on an LSF machine, run it on the head node with

```
R --vanilla --args RBATCH lsf < param-sim.R
```

This then batches the following commands, also given as output from running the code:

```
bsub -q normal "R --vanilla --args  seed 1000 n 25 mu ""c(0,0.1)""
  < mainSim.R > mainSim.Rout1000"
bsub -q normal "R --vanilla --args  seed 1001 n 25 mu ""c(0,0.2)""
  < mainSim.R > mainSim.Rout1001"
...
bsub -q normal "R --vanilla --args  seed 1009 n 25 mu ""c(0,1)""
  < mainSim.R > mainSim.Rout1009"
```

To run it on a MOSIX cluster, run it on a node (or the node you have designated as the "head" node) with:

```
R --vanilla --args RBATCH mosix < param-sim.R
```

Which runs the commands

```
nohup mosrun -e -b -q R --vanilla --args seed 1000 n 25 mu "c(0,0.1)"
  < mainSim.R > mainSim.Rout1000 &
...
```

Lastly, if you have a different cluster platform, or want to override the default queues there are a few other options you can set. The option `BATCH` controls the batch submission string. For example, setting it to be `"bsub -q long"` would choose a different queue on the LSF platform. Sometimes an "`&`" character is needed after the cluster submission character. The option `QUOTE` controls if a command string should be quoted (`'""'` for LSF, `""` for MOSIX), and `ARGQUOTE` controls if each argument should have a special quotation mark when they are vectors or strings with spaces in them (`'"""'` for LSF, `'"'` for MOSIX). Finally `RUN` controls if a statement is run (`RUN 1`) versus just the command string that would be run is printed for debugging (`RUN 0`).

Alternatively, if one is sure that it is going to run on one particular cluster, then, for LSF, one could replace the line above to read

```
seed <- rbatch("sim.R", seed = seed, n = 25, mu = c(0, i / 10),
  rbatch.control = rbatch.lsf())
```

One could also use `rbatch.mosix()` for MOSIX. For custom clusters, one would use `rbatch.lsf` or `rbatch.mosix` and alter the default parameters `BATCH`, `BATCHPOST`, `QUOTE`, `ARGQUOTE`, `RUN`, and `MULTIPLIER`. All of these arguments behave as described previously.

After this is done running, 10 CSV files will be created. These can then be joined by running the following command in R after loading in the **batch** package

```
R> mergeCsv()
```

The merged results will be by default put in the file '`allResults.csv`', unless the `outfile` argument to the `mergeCsv` function is set to a different value indicating where it should go. This file can be opened with any spreadsheet program. Note that this function will attempt to merge all CSV files in the current working directory. So one should make sure there are none

there before running the `rbatch` routine, or instead save all of the CSV files into a separate output directory.

3.3. Splitting into multiple pieces

Now suppose that we are running '`sim.R`', but now the issue is that the simulation is taking too long to run. We can split up our simulation into pieces. We can create the following code in '`split-sim.R`':

```
library("batch")
seed <- rbatch("sim.R", seed = 1000, nsim = 1000)
rbatch.local.run()
```

Note that we have lowered the number of simulations of each job by 10 fold. Now, to make up for that loss, we run the simulation 10 times with the command line argument `MULTIPLIER`. That is:

```
R --vanilla --args MULTIPLIER 10 < split-sim.R
```

It is for this reason that the code above stores a new value for `seed` from the `rbatch` function. When the multiplier is set, the `rbatch` function will automatically increments the seed for each execution, and then passes the next seed to use back. So when `MULTIPLIER` is set to 10, the first simulation is set to seed 1000, the second to 1001, and the value of `seed` returned from the execution of `rbatch` is 1010. The net effect of all of this is the same number of simulations, split in parallel.

Finally, to paste these results together, one can run from R after loading in **batch**:

```
R> mergeCsv(every = 10)
```

Here the CSV file is averaged for every set of 10 lines (in accordance with the `MULTIPLIER` value) to produce a combined output. So long as you follow the convention of having only one row of output in each output CSV file, this function should work properly. Otherwise you can still use the function with the default `every = 1`, but you will have to merge the resulting CSV file manually.

4. Auxiliary functions

One other function is provided in the R package to help with splitting thing, so they can be run across a cluster. The function `msplit` can take a vector and break it up into K pieces of near equal size. This is returned as a list of vectors. For instance if we wanted to split SNP1, SNP2, ..., SNP10 into 3 groups for processing with the `rbatch` function, we would run the following

```
R> msplit(paste("SNP", 1:10, sep = ""), 3)
[[1]]
[1] "SNP1" "SNP2" "SNP3" "SNP4"
```

```
[[2]]
[1] "SNP5" "SNP6" "SNP7"

[[3]]
[1] "SNP8"  "SNP9"  "SNP10"
```

5. Discussion

We have presented the R package `batch` that easily allows a user to specifiy command line options to R script files. This allows for an easy parallelization of code, with many helper functions also provided with the code. With no modification to your batch R script, you can run your code on your local machine, utilizing all the CPUs available, or on a cluster if available. Future work is to support more default arguments for cluster specifications besides LSF and MOSIX.

The package allows for easy parallelization on different clusters and your local machine by altering command line options to the batch submission script file. One suggestion is to create an alias to the R batch submission for each cluster and a local non-Windows machine, so that it is the same on every machine you use. For example on a local non-Windows machine, you might add the following line to the '`.bashrc`' file in your home directory:

```
alias rbatch = 'R --vanilla --args'
```

Then, on your LSF cluster, you could instead add the line

```
alias rbatch = 'R --vanilla --args RBATCH lsf'
```

and on your MOSIX cluster,

```
alias rbatch = 'R --vanilla --args RBATCH mosix'
```

This would also make it trivial if you had any other cluster implementation; you simply need to create the proper alias once. Then the above code

```
R --vanilla --args MULTIPLIER 10 < split-sim.R
```

would instead be run on any of your machines/clusters with

```
rbatch MULTIPLIER 10 < split-sim.R
```

Then there is absolutely no difference in submitting jobs locally as on a cluster. One could further modify this with the option `RBATCH` to have different aliases for different queues; for example, on LSF

```
alias rbatch-long = 'R --vanilla --args RBATCH lsf BATCH "bsub -q long"'
```

or in MOSIX with

```
alias rbatch-q25 = 'R --vanilla --args RBATCH mosix BATCH
  "nohup mosrun -e -b -q25"'
```

Acknowledgments

Funding was provided by NIH Award #R25CA112355.

References

Barak A, La'adan O (1997). "The MOSIX Multicomputer Operating System for High Performance Cluster Computing." *Future Generations in Computer Systems*, **13**(4), 361–372.

Burns G, Daoud R, Vaigl J (1994). "LAM: An Open Cluster Environment for MPI." In *Proceedings of Supercomputing Symposium*, pp. 379–386. URL http://www.lam-mpi.org/.

Eddelbuettel D (2010). *CRAN Task View: High-Performance and Parallel Computing with R*. Version 2010-12-12, URL http://CRAN.R-project.org/view=HighPerformanceComputing.

Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS (2004). "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation." In *In Proceedings, 11th European PVM/MPI Users' Group Meeting*, pp. 97–104.

Gropp W, Lusk E, Doss N, Skjellum A (1996). "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard." *Parallel Computing*, **22**(6), 789–828.

Knaus J, Porzelius C, Binder H, Schwarzer G (2009). "Easier Parallel Computing in R with **snowfall** and **sfCluster**." *The R Journal*, **1**, 54–59.

Platform Computing Corporation (2001). "Platform LSF: The HPC Workload Management Standard." URL http://www.platform.com/.

R Development Core Team (2010). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

REvolution Computing (2009). "**foreach**: Foreach Looping Construct for R." R package version 1.3.0, URL http://CRAN.R-project.org/package=foreach.

Rossini AJ, Tierney L, Li N (2007). "Simple Parallel Statistical Computing in R." *Journal of Computational and Graphical Statistics*, **16**(2), 399–420.

Schmidberger M, Morgan M, Eddelbuettel D, Yu H, Tierney L, Mansmann U (2009). "State of the Art in Parallel Computing with R." *Journal of Statistical Software*, **31**(1), 1–27.

Smith C, Warnes G, Kuhn M, Coulter N (2007). "**Rlsf**: Interface to the LSF Queuing System." R Package. URL http://cran.r-project.org/web/packages/Rlsf/index.html.

Squyres JM, Lumsdaine A (2003). "A Component Architecture for LAM/MPI." In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pp. 379–387. Springer-Verlag, Venice, Italy.

Urbanek S (2011). "**multicore**: Parallel Processing of R Code on Machines with Multiple Cores or CPUs." R package version 0.1-4, URL http://CRAN.R-project.org/package=multicore.

Warnes GR (2009). "**fork**: R Functions for Handling Multiple Processes." R package version 1.2.2, URL http://CRAN.R-project.org/package=fork.

Yu H (2002). "**Rmpi**: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14.

Affiliation:

Thomas J. Hoffmann
Department of Epidemiology and Biostatistics
University of California, San Francisco
San Francisco, CA 94107, United States of America
E-mail: tjhoffm@gmail.com
URL: http://sites.google.com/site/thomashoffmannproject/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 39, Code Snippet 1 *Submitted:* 2009-12-16
March 2011 *Accepted:* 2010-08-09