Efficient *R* Programming

Martin Morgan

Fred Hutchinson Cancer Research Center
Seattle, WA, USA

July 30 2010

1 Pitfalls

These brief exercises are meant to illustrate some common obstacles to efficient *R* programming. The idea is that you'll follow along with the text, evaluating the instructions in your own *R* session.

The basic scenario is a genome-wide association study. There are 1000 individuals. Case versus control status, gender, and age were recorded for each, along with genotype at 10000 SNPs. The data is entirely synthetic.

A suggestion from the lectures is to investigate packages that might already efficiently implement the calculations you are trying to perform. Check out the *snpMatrix* package after doing these exercises. How much of the following does this package make unnecessary? Hint: almost all of it!

1.1 Getting going

To begin:

1. Start an *R* session

2. Load the package for this conference.

   ```
   > library(EfficientR)
   ```

3. Read several pre-defined functions in to the *R* session

   ```
   > fl <- system.file("script", "pitfalls.R",
   +                   package="EfficientR")
   > source(fl)
   ```

1.2 Basic performance measurement and data I/O

The first activities are meant to illustrate the use of `system.time` to evaluate performance, `object.size` to investigate how much memory an object uses, and `identical` and `all.equal` to compare objects. We also look at how to more effectively read data in to *R*.

First, let's take a look at `fname0`, a file path to the GWAS data set, and the function `f0`. The code below defines `fname`; `f0` is defined in the `pitfalls.R` file. Here are the definitions, and the result assigned to the variable `gwas1`

```
> fname0 <- system.file("extdata", "gwas_2.csv",
+                        package="EfficientR")
> f0

function(fileName) {
    read.csv(fileName, row.names=1)
}

> gwas1 <- f0(fname0)
```

This function simply reads in the file. Let's use `system.time` to evaluate how long this takes

```
> system.time(gwas0 <-  f0(fname0))

   user  system elapsed
  7.519   0.145   7.856

> dim(gwas0)

[1]  1000 10003
```

Note the way in which the `<-` assignment to variable `gwas0` occurs in `system.time`. See the help page `?system.time` to understand the output; we're usually interested in the `user.time`.

The function `f0` does its job, but perhaps we can improve on it. For instance, suppose we were interested in a preliminary investigation, and in particular reading just the case/control status, sex, and age of the samples. The following shows two different functions that allow us to read in just the information we're interested in

```
> f1

function(fileName) {
    colClasses <- c("character", "factor", "factor", "integer",
                    rep("NULL", 10000))
    read.csv(fileName, row.names=1, colClasses=colClasses)
}

> f2
```

```
function(fileName) {
    what <- c(list(id=character(),
                   CaseControl=character(),
                   Sex=character(),
                   Age=integer()),
              rep(list(NULL), 10000))
    input <- scan(fileName, what=what, sep=",", skip=1)
    as.data.frame(input[2:4], row.names=input[[1]])
}
```

Compare these functions with each other and `f0`, and with the relevant help pages `?read.csv`, `scan` to identify the key steps for efficient data input. Now let's see how they perform in terms of evaluation time:

```
> system.time(gwas1 <- f1(fname0))

   user  system elapsed
  4.084   0.066   4.165

> system.time(gwas2 <- f2(fname0))

   user  system elapsed
  2.210   0.059   2.281
```

Since we've read in just the data we're interested in, we should have saved quite a bit of space

```
> object.size(gwas0)

43661544 bytes

> object.size(gwas1)

49064 bytes
```

The functions `f1` and `f2` are meant to return the same result, just implemented in slightly different ways. We can verify this with `identical`

```
> identical(gwas1, gwas2)

[1] TRUE
```

Having read in the data, we might be interested in summarizing the phenotypes we have, e.g.,

```
> xtabs(~CaseControl + Sex, gwas2)

           Sex
CaseControl   F   M
    Case    248 252
    Control 266 234
```

1.3 Character manipulation

Here we'll look at one aspect of *R* that can sometimes have a surprising performance penalty and sometimes tricky semantics: character manipulation. We'll learn some additional techniques for monitoring performance, as well as gain an appreciation for the benefits of appropriate function choice.

Let's look at the genotypic data only, by dropping the first three columns from the data frame; we'll take a peak at the first six rows (`head`) of the first five columns of the genotype information. (Pay no attention to the `names<-` call; it makes the column names more distinct and is useful later).

```
> gtype <- gwas0[,-(1:3)]
> names(gtype) <- paste("snp", names(gtype), sep="_")
> head(gtype[,1:4])

     snp_X1 snp_X2 snp_X3 snp_X4
id_1     AA     AB     AA     AB
id_2     AA     AA     AA     AA
id_3     AA     AA     AA     AA
id_4     AB     AA     AB     AA
id_5     AB     AA     AA     AA
id_6     AB     AB     AA     AA
```

Note that the rows have names (e.g., `id_1`).

A function `shuffle0` might come in handy if one were wanting to randomize genotypes, and to return the randomized genotypes as a matrix.

```
> shuffle0

function(genotypes, seed=123L) {
    set.seed(seed)
    samp <- sample(genotypes)
    g <- unlist(samp)
    matrix(g, ncol=ncol(genotypes))
}
```

We use `seed` to make the results reproducible across invocations. `sample(genotypes)` permutes the columns of the `gtype` data frame. The `unlist` and `matrix` commands are meant as a first attempt at creating a matrix from our permuted data frame – we 'collapse' the sampled genotypes into a vector, and then shape the vector into a matrix. Let's measure how long this takes:

```
> system.time(s0 <- shuffle0(gtype))

   user  system elapsed
 42.705   1.045  44.514
```

Wow, that seems like a fairly long time for a relatively simple set of operations. I wonder what's going on?

```
> profFile <- tempfile()
> Rprof(profFile)      # start gathering profile information
> s0 <- shuffle0(gtype)
> Rprof()              # stop
> head(summaryRprof(profFile)$by.self)

            self.time self.pct
"unlist"         9.72     67.1
"structure"      1.40      9.7
"unclass"        0.72      5.0
"as.vector"      0.62      4.3
"levels"         0.58      4.0
"match"          0.52      3.6
            total.time total.pct
"unlist"         13.40      92.5
"structure"       1.52      10.5
"unclass"         0.72       5.0
"as.vector"       0.62       4.3
"levels"          0.62       4.3
"match"           0.52       3.6
```

A large fraction of the time is spent on the `unlist` function. A little experimentation suggests what the problem might be:

```
> gsubset <- gtype[1:2, 1:3]
> unlist(gsubset)

snp_X11 snp_X12 snp_X21 snp_X22 snp_X31 
     AA      AA      AB      AA      AA 
snp_X32 
     AA 
Levels: AA AB BB
```

Notice that the value of `unlist` is a vector with names, and that the names have been constructed to be unique. We can reference the help page `?unlist`, and arrive at a better solution `shuffle1` that avoids creating names.

```
> shuffle1

function(genotypes, seed=123L) {
    set.seed(seed)
    samp <- sample(genotypes)
    g <- unlist(samp, use.names=FALSE)
    matrix(g, ncol=ncol(genotypes))
}
```

This almost trivial change has a big influence on performance, without changing the result:

```
> system.time(s1 <- shuffle1(gtype))

   user  system elapsed
  1.749   0.452   2.214

> identical(s0, s1)

[1] TRUE
```

Finally, in *R* it is common to be able to 'cast' from one data structure to another. `shuffle2` does this

```
> shuffle2

function(genotypes, seed=123L) {
    set.seed(seed)
    as.matrix(sample(genotypes))
}

> system.time(s2 <- shuffle2(gtype))

   user  system elapsed
  1.686   0.090   1.783
```

Note that the performance of `as.matrix` is comparable to our `shuffle1`. Are the results the same?

```
> identical(s1, s2)

[1] FALSE
```

Oh oh! This doesn't look good. But maybe it's just that our results `s1` do not have `dimnames`, whereas `s2` might?

```
> all.equal(s1, s2)

[1] "Attributes: < Length mismatch: comparison on first 1 components >"

> all.equal(s1, s2, check.attributes=FALSE)

[1] TRUE
```

(The result of the first call to `all.equal` is fairly cryptic; this is, unfortunately, typical). The built-in function `as.matrix` is performing as well as our version, and is doing a better job of tracking important information (row and column names) through the analysis.

1.4 Preparing for an analysis

To close this first exercise, let's set up a fairly typical but statistically advanced GWAS-style analysis. I gloss over the details, but we are doing a regression, where we try to explain the case versus control classification in terms of sex, age, and a single SNP. We'd ultimately be interested in SNPs that were most effective at classifying a sample into case or control. Here's a function that might perform the analysis of one SNP:

```
> snp0

function(i, gwas) {
    snp <- gwas[,-c(1:3)]
    glm(CaseControl ~ Age + Sex + snp[,i], family=binomial, data=gwas)
}

> snp0(1, gwas0)

Call:  glm(formula = CaseControl ~ Age + Sex + snp[, i], family = binomial,      data = gwas

Coefficients:
(Intercept)          Age         SexM
  -0.159373     0.005783    -0.143788
 snp[, i]AB   snp[, i]BB
  -0.004749    -0.020283

Degrees of Freedom: 999 Total (i.e. Null);  995 Residual
Null Deviance:           1386
Residual Deviance: 1385         AIC: 1395
```

These computations are expensive, especially when we remember that we have (in real examples) hundreds of thousands of SNPs

```
> system.time(result <- lapply(1:10, snp0, gwas0))

   user  system elapsed
  4.648   0.308   4.979
```

We'll see some approaches to getting better throughput in the next section.

2 Data I/O: Streaming

We continue with the scenario of a genome-wide association study. There are 1000 individuals. Case versus control status, gender, and age were recorded for each, along with genotype at 10000 SNPs. The data is entirely synthetic.

The following defines a function `fapply.csv` that (tries to!) stream data from a file through a function, much like `lapply` steams elements of a list through a function.

```
> readScript("fapply.R")

 [1] fapply.csv <-
 [2]     function(fname, FUN, ..., nrows=200L, header=TRUE,
 [3]              colClasses=NA, col.names, .reduce)
 [4] {
 [5]     conn <- file(fname, open="r")
 [6]     ## first chunk; special: remember header info
 [7]     chunk <- read.csv(conn, header=header, ..., nrows=nrows,
 [8]                       colClasses=colClasses)
 [9]     colClasses <- sapply(chunk, class)
[10]     colNames <- c("", names(chunk))
[11]     result <- list(); it <- 1
[12]     repeat {
[13]         result[[it]] <- FUN(chunk, ...)
[14]         if (nrow(chunk) != nrows) break
[15]         chunk <- read.csv(conn, header=FALSE, ...,
[16]                           nrows=nrows, colClasses=colClasses,
[17]                           col.names=colNames)
[18]         if (nrow(chunk) == 0L) break
[19]         it <- it + 1
[20]     }
[21]     close(conn)
[22]     if (missing(.reduce)) result
[23]     else .reduce(result)
[24] }

> fl <- system.file("script", "fapply.R",
+                   package="EfficientR")
> source(fl)
```

By default, the function takes a file name, a function to be applied to each chunk, and parameters influencing how the chunks are processed. Let's give it a whirl, by reading the GWAS file in chunks of 200 rows at a time. For each chunk, we'll calculate the fraction of heterozygous loci. Here's our function

```
> hetero <- function(chunk, ...) {
+   cat("starting chunk\n")
+   rowSums(chunk[,-(1:3)]=="AB") / (ncol(chunk) - 3)
+ }
```

A common operation, both in stream processing and in distribution of tasks for parallel evaluation, is to 'reduce' the results from different chunks / tasks into a single meaningful object. The `.reduce` argument to `fapply.csv` is meant to be a function that performs the reduction. It expects a list, with each element of the list the result of the function operating on a chunk, e.g., each element being the result of `hetero`. As a simple reduce function, we'll use `unlist`. The result

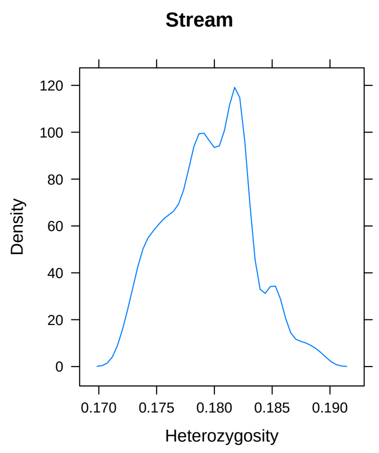

Figure 1: Individual heterozygosity, 'stream' processing

should be a vector of heterozygosities that we can manipulate or visualize in the usual way. Let's give this a go...

```
> res <- fapply.csv(fname0, hetero, row.names=1, .reduce=unlist)
```

```
starting chunk
starting chunk
starting chunk
starting chunk
starting chunk
```

```
> length(res)
```

```
[1] 1000
```

```
> library(lattice)
> print(densityplot(res, plot.points=FALSE, xlab="Heterozygosity",
+                   main="Stream"))
```

As exercises:

1. Explore different functions that might usefully be used in a streaming context. For instance, suppose the goal is to calculate heterozygosities for each SNP, rather than individual. What might the `FUN` and `.reduce` arguments to `fapply.csv` look like?

2. As an advanced exercise, download and investigate the *biglm* package for fitting linear models in a streaming fashion. When (what types of model and data) would the *biglm* package be particularly useful?

3. As another advanced exercise, consider how `fapply.csv` might be modified to work in parallel.

3 SQL

Let's open the GWAS phenotype data from its SQLite data base…

```
> library(RSQLite)
> db0 <- sub("csv$", "sql", fname0) # $
> drv <- dbDriver("SQLite")
> conn <- dbConnect(drv, dbname=db0)
> dbListTables(conn)
```

```
[1] "gwasPhenotypes"
```

…and read all data into a data frame

```
> q <- dbSendQuery(conn, "SELECT * FROM gwasPhenotypes")
> df <- fetch(q, n=-1)
> clear <- dbClearResult(q)
> head(df)
```

```
  row_names CaseControl Sex Age
1      id_1        Case   M  40
2      id_2        Case   F  33
3      id_3        Case   F  40
4      id_4        Case   F  38
5      id_5        Case   M  42
6      id_6        Case   F  39
```

1. How does the query result `df` differ from the original data used to create the data base? Look carefully at the structure of the data frame, data values, and the class of columns in the data frame. Does `all.equal` provide additional insight into differences or similarities? The original data can be retrieved with

   ```
   > gwasPhenotype <- f2(fname0)
   ```

2. What's the role of each of the commands starting with `db` in the code above? use the help pages for assistance.

3. Two other ways of processing an entire table are

   ```
   > df1 <- dbGetQuery(conn, "SELECT * FROM gwasPhenotypes")
   > df2 <- dbReadTable(conn, "gwasPhenotypes")
   ```

4. SQL allows for more complicated queries. For instance evaluate the following and speculate on the meaning of the SQL components.

   ```
   > df <- dbGetQuery(conn,
   +                  "SELECT age FROM gwasPhenotypes
   +                   WHERE sex = 'F'")
   ```

5. SQL allows selection of several fields by providing a comma-delimiting string after 'select'. Can you formulate a SQL query to select the row names and `Age` of all females?

6. SQL allows for functions to replace variable names as the 'select' component. Verify (e.g., by comparison with the equivalent R statements on the entire data set) that the following returns the number of females over 40 in the data set.

   ```
   > dbGetQuery(conn,
   +            "SELECT COUNT(*) FROM gwasPhenotypes
   +             WHERE age > 40 AND sex = 'F'")

     COUNT(*)
   1      233
   ```

Don't forget to tidy up when done!

```
> ok <- dbDisconnect(conn)
```

4 NetCDF

Turn now to the NetCDF file.

```
> ncdf0 <- sub("csv$", "nc", fname0) #$
> library(ncdf)
> nc <- open.ncdf(ncdf0)
> nc

[1] "file /var/folders/3d/3dke84UEF3iPUjKHPimSRU+++TM/-Tmp-//Rinst235580085/EfficientR/extda
[1] "Sample   Size: 1000"
[1] "SNP   Size: 10000"
[1] "------------------------"
[1] "file /var/folders/3d/3dke84UEF3iPUjKHPimSRU+++TM/-Tmp-//Rinst235580085/EfficientR/extda
[1] "int Genotype[Sample,SNP]  Longname:Genotype Missval:-1"
```

The `nc` object contains useful summary information. It is in a list-like structure. For instance, we can discover the number of dimensions and their lengths with commands like

```
> names(nc)

 [1] "id"           "ndims"
 [3] "natts"        "unlimdimid"
 [5] "filename"     "varid2Rindex"
 [7] "writable"     "dim"
 [9] "nvars"        "var"
```

```
> names(nc[["dim"]])
```

```
[1] "Sample" "SNP"
```

```
> names(nc[["dim"]][["SNP"]])
```

```
[1] "name"          "len"
[3] "unlim"         "id"
[5] "dimvarid"      "units"
[7] "vals"          "create_dimvar"
```

```
> nSnps <- nc[["dim"]][["SNP"]][["len"]]
```

1. What information can you 'discover' about the `Gentoype` variable?
2. Get the first 100 individuals and their SNPs with

   ```
   > g <- get.var.ncdf(nc, "Genotype", start=c(1, 1), count=c(10, nSnps))
   ```

3. From the examples on `?get.var.ncdf`, is there another way to discover the dimensions of variables, and an easy way to retrieve all values in a dimension?
4. How would you retrieve samples 501 to 600, SNPs 1000 to 2000?
5. Would it be possible to retrieve all SNPs from females? What if the information from the *RSQLite* data were available? What kind of work-around might you imagine?

Remember to tidy up when done!

```
> ok <- close(nc)
```

5 *R* on Clusters: *Rmpi*

These exercises require access to a cluster and so cannot be completed in this session; they may provide a useful resource. Commands necessary to run *R* on a cluster tend to be very particular to each system; the following uses `slurm` to manage cluster jobs. A somewhat easier approach is to use multiple cores on a single machine. The *multicore* and *foreach* packages provide one example of this.

The basic scenario for these exercises is a genome-wide association study. There are 1000 individuals. Case versus control status, gender, and age were recorded for each, along with genotype at 10000 SNPs. The data is entirely synthetic.

We've seen how to read in the data, and we're at the stage where we'd like to fit generalized linear models to each SNP. We'll get to the point where we can evaluate the model in parallel, and leave for an 'advanced exercise' steps necessary to make this actually a computationally feasible endeavor.

5.1 Getting going

We start by becoming familiar with submitting batch jobs, and exploring a common model for parallel evaluation in R.

Let's start by reviewing and submitting a batch job. Here's the content of the file:

```
> readScript("spawn.R")

 [1] # salloc -N 4 orterun -n 1 R -f spawn.R
 [2] library(Rmpi)
 [3] nWorkers <- mpi.universe.size() - 1L
 [4] mpi.spawn.Rslaves(nsl=nWorkers)
 [5] mpi.remote.exec(mpi.comm.rank())
 [6] mpiRank <- function(i)
 [7]   c(i=i, rank=mpi.comm.rank())
 [8] mpi.parSapply(seq_len(2*nWorkers), mpiRank)
 [9] mpi.close.Rslaves()
[10] mpi.quit()
[11]
```

- Line 1 is a reminder about how we'll run this job.
- Line 2 loads the *Rmpi* library. `mpi` (message passing interface) is a standard way to write parallel programs; *Rmpi* is a *wrapper* around the `mpi` libraries.
- Line 3 consults the environment we are running in to find out how many computer nodes there are available for this process. We'll use one of the nodes for the 'manager', the others for the 'workers' (*Rmpi* uses the terminology 'master' and 'slave').
- Line 4 starts the workers. Each worker starts on a particular machine, determined when we submit the batch job. The workers are assigned a `rank` (0 for the manager, 1, 2, ...for the workers); we'll mention `comm` latter. The worker starts a regular session of R, consuming just as many resources and with access to the same facilities as the manager.
- Line 5 is an example of a *Rmpi* command. It evaluates the expression `mpi.comm.rank()` on each of the workers, and returns the result to the R session. The expression `mpi.comm.rank()` returns the rank of the machine on which it is evaluated; we should get one value for each worker.
- Lines 6-7 define a function that takes a single argument, and returns that argument and the rank of the worker on which the function was evaluated.
- Line 8 introduces a very useful way of using *Rmpi*: `mpi.parSapply` is like `sapply`, except that the calculation is distributed (as evenly as possible, by default) across the workers. So this line distributes `1:(2*nWorkers)` 'tasks'

(i.e., determining `mpiRank`) across each worker. Other useful high-level functions include `mpi.parLapply` and `mpi.parReplicate`.

- Lines 9 and 10 are 'nice' ways to clean up, but are not actually required in the environment we're running in.

OK, this is how the job submission might look:

```
% salloc -N 4 mpirun -n 1 R --quiet -f spawn.R
```

There are three parts to the job submission. First we use `slurm`, the 'job controller', to allocate 4 computer nodes for our job. Then we use `mpirun` to launch, within the `slurm` allocation, a single process. The process is *R*, and we launch *R* in such a way that it is quiet (e.g., no start-up message) and takes its input from the file `spawn.R`. `slurm` responds by granting our allocation, launching `mpirun` and then R, and then echoing the output back to the screen:

```
salloc: Granted job allocation 239858
> # salloc -N 4 orterun -n 1 R -f spawn.R
> library(Rmpi)
> nWorkers <- mpi.universe.size() - 1L
> mpi.spawn.Rslaves()
        4 slaves are spawned successfully. 0 failed.
master (rank 0, comm 1) of size 5 is running on: hyraxB69
slave1 (rank 1, comm 1) of size 5 is running on: hyraxB69
slave2 (rank 2, comm 1) of size 5 is running on: hyraxB70
slave3 (rank 3, comm 1) of size 5 is running on: hyraxB71
slave4 (rank 4, comm 1) of size 5 is running on: hyraxB72
> mpi.remote.exec(mpi.comm.rank())
  X1 X2 X3 X4
1  1  2  3  4
> mpiRank <- function(i)
+   c(i=i, rank=mpi.comm.rank())
> mpi.parSapply(seq_len(2*nWorkers), mpiRank)
     [,1] [,2] [,3] [,4] [,5] [,6]
i       1    2    3    4    5    6
rank    1    2    2    3    3    4
> mpi.close.Rslaves()
[1] 1
> mpi.quit()
salloc: Relinquishing job allocation 239858
```

We might capture the output by running the R script in 'batch' mode

```
% salloc -N 4 mpirun -n 1 R CMD BATCH spawn.R
```

which would generate a file `spawn.Rout` in our current directory. Another possibility is that we use standard Linux redirection and piping to manipulate the output, e.g.,

```
% salloc -N 4 mpirun -n 1 R --vanilla < spawn.R | wc
```

As exercises

1. Repeat the above actions in your session.
2. Try changing the number of nodes allocated by `salloc` (within reason!)
3. Investigate the `salloc` help page (`man salloc`, using 'f', 'b' to page forward or backward, 'q' to quit). Can you determine the difference between the -N and -n options?
4. Look at the scripts `random.R` and `simd.R` (use `readScript("random.R")` and `readScript("simd.R")`). Can you tell, from consulting with the class notes, what these scripts do?

5.2 Catching up

In the first exercise, we set up a statistically advanced GWAS-style analysis. We are doing a regression, where we try to explain the case versus control classification in terms of sex, age, and a single SNP, using a generalized linear model. We can use this on a single node by reading in the data and doing a number of analyses. Here's a simple script:

```
> readScript("snp0.R")

 [1] system("hostname")
 [2] fname0 <- "/home/mtmorgan/hpc/extdata/gwas_2.csv"
 [3] snp0 <- function(i, gwas) {
 [4]     snps <- gwas[,-(1:3)]
 [5]     glm(CaseControl ~ Age + Sex + snps[,i], family=binomial,
 [6]         data=gwas)$coef
 [7] }
 [8]
 [9] gwas <- read.csv(fname0, row.names=1)
[10] result <- lapply(1:20, snp0, gwas)
[11] result[[1]]
```

We can invoke this on a single node in several ways; here are two

```
srun R -f snp0.R
salloc -N 1 mpirun -n 1 R -f snp0.R
```

Now let's modify this script to support parallel evaluation. Here's a first pass at the function:

```
> readScript("snp2.R")
```

```
 [1] library(Rmpi)
 [2] mpi.spawn.Rslaves()
 [3]
 [4] ## load gwas on each node
 [5] mpi.remote.exec({
 [6]     fname0 <- "/home/mtmorgan/hpc/extdata/gwas_2.csv"
 [7]     gwas <<- read.csv(fname0, row.names=1)
 [8] }, ret=FALSE)
 [9]
[10] ## evaluate in parallel
[11] snp0 <- function(i) {
[12]     snps <- gwas[,-(1:3)] ## global reference 'gwas'
[13]     glm(CaseControl ~ Age + Sex + snps[,i], family=binomial,
[14]         data=gwas)$coef
[15] }
[16] system.time(result <- mpi.parLapply(1:200, snp0))
[17]
[18] result[[1]]
[19]
[20] mpi.close.Rslaves()
[21] mpi.quit()
```

The key features are:

- Lines 1 and 2: load *Rmpi* and spawn workers.
- Lines 5-8: load the `gwas` data set on each node, assigning to a global variable (with ■-).
- Lines 11-15: define the function for fitting a single SNP, referencing the global variable. Return only the coefficients, rather than entire model.
- Line 16: perform evaluation in parallel.

Note how we use the shared drives of the cluster to 'distribute' the data to each node, and restrict the amount of data that is returned to the fitted coefficients (`glm` returns the original data along with the fit, and so is large). One motivation for taking these steps is to reduce the amount of explicit communication between nodes.

We can readily evaluate our script on 2, 5, or 10 nodes as, e.g.,

```
salloc -N 2 mpirun -n 1 R -f snp2.R
salloc -N 5 mpirun -n 1 R -f snp2.R
salloc -N 10 mpirun -n 1 R -f snp2.R
```

5.3 Looking forward

The solution in `snp2.R` shows how to 'get the job done', but it is not very elegant, the code in `snp0.R` had to be modified fairly extensively, and there are a number of redundant calculations. As an advanced exercise,

1. The 'ugliest' parts of the script are the use of a global variable `gwas` to store the SNPs on each node, and the separation of the 'data distribution' stage from the analysis stage. These are problems because they make the script more fragile and difficult to understand. Can you revise this script to address these issues? What about using a pattern like that in the `simd.R` (use `readScript("simd.R")`)?

2. For those with a strong statistical background.

   One of the expensive parts of `glm` is the construction of a design matrix. Note that most of the design matrix is constant across SNPs. Use this observation to construct a design matrix that excludes SNPs once, and explore how to modify this when a single SNP is added to the model.

   The `glm` algorithm depends on good starting values. Use this and the idea that the effect of individual SNPs is usually small to calculate starting values based on just the phenotypic data.

   Scanning all SNPs is likely a 'first pass' at data analysis – are there any SNPs that stand out as particularly interesting? `glm` is an iterative algorithm. What useful information can be obtained if the algorithm is restricted to just a single iteration?

As an advanced exercise that integrates the data access and parallel portions of the lab, here's a script that spawns two MPI nodes to process (e.g., calculate heterozygosity per sample, `nchetero`) different components of the data. Can you identify the 'reduce' functionality to transform the results of the separate tasks into a single object? Can you transform the script below into the 'SIMD' style of MPI introduced last week, including making the code scalable to different numbers of nodes?

```
> nSamples <- 1000
> nWorkers <- 2
> ## divide samples between workers; .splitIndices in Rmpi
> library(Rmpi)
> idx <- lapply(.splitIndices(nSamples, nWorkers), range)
> ## calculate individual heterozygosity for samples in idxElt
> nchetero <- function(idxElt, ...)
+ {
+     snpN <- nc[["dim"]][["SNP"]][["len"]]
+     start <- c(idxElt[1], 1)
+     count <- c(diff(idxElt) + 1L, snpN)
+     d <- get.var.ncdf(nc, "Genotype", start=start, count=count)
+     rowSums(d==2) / ncol(d)
+ }
> mpi.spawn.Rslaves(nsl=nWorkers)
> mpi.bcast.cmd(library(ncdf))
> mpi.bcast.Robj2slave(ncdf0)
> mpi.bcast.cmd(nc <- open.ncdf(ncdf0))
> res <- mpi.parLapply(idx, nchetero)
> mpi.bcast.cmd(close(nc))
> ok <- mpi.close.Rslaves()

> densityplot(unlist(res), plot.points=FALSE, xlab="Heterozygosity",
+             main="MPI and NetCDF")
```

Notice that this last example gains two speed-ups: I/O from use of binary NetCDF files, and distributed computation across nodes.