Parallel Computing in R

Xie Chao and Tan Tin Wee

May 5, 2010

Contents

1 Introduction 1
1.1 Background 1
1.2 Prerequisite 2

2 Parallel Computing in R 2
2.1 The Scenario 2
2.2 Multicore Computers 6
2.3 Simple Network Of Workstations (snow) 6
2.4 Linux Cluster 9
2.5 *"One Ring to Rule Them All"* 10
2.5.1 The foreach Framework 10
2.5.2 Register a Parallel Backend 11
2.5.3 Serial Code 11
2.5.4 doMC 12
2.5.5 doSNOW 13
2.5.6 doMPI 13

3 Parallel R in Cloud 14

4 Useful Resources 14

1 Introduction

1.1 Background

- Data becomes cheaper, because machines are becoming cheaper and faster.
- Computers become cheaper and faster too. However, the performance of single processor core has not changed much recently — instead we are getting multi-core processor, and clusters of multi-cores.
- To handle the large amount of data collected from modern machines with large amount of computing units, we need to use parallel or distributed computing,

in other words, to get many computers to perform the same task simultaneously.

- Many problems are "embarrassingly parallel". In other words, they can be split into many smaller tasks and passed on to many many computers for the computation to be carried out simultaneously .
- For this to be achieved, we need to have a way for identifying which components can be distributed and processed in a parallel manner, and how the data and the computation can be distributed, and how the output can be collected from all these computers, and reassembled.
- R is a programming language and comes with a software environment fully enabled for statistical computing and graphics. It is a *de facto* standard programming language for statistics and has strong support for bioinformatics and computational biology, where statistical procedures are increasingly more frequently used for biological data analysis.

1.2 Prerequisite

This tutorial on parallel and distributed computing for bioinformatics applications in R requires a basic understanding of R. If you do not have any experience with R, we suggest you to read the manual *An Introduction to R* at the R-project home page (http://cran.r-project.org/doc/manuals/R-intro.html) or the tutorial at http://www.statmethods.net/stats/.

For enabling parallel computing in R, we need several add-on packages, which can be installed as follows:

```
$R
> install.packages(c("multicore", "snow", "foreach", "doMC", "doSNOW"))
> install.packages(c("Rmpi", "doMPI")) # only if on a cluster with MPI support
> source("http://bioconductor.org/biocLite.R")
> biocLite("Biobase")
```

2 Parallel Computing in R

2.1 The Scenario

Let's use the sample data set geneData from the Biobase package in Bioconductor. This data set contains expression data for 500 genes (rows) in 26 samples (columns).

```
> data(geneData, package = "Biobase")   # load the data
> dim(geneData)                         # check indeed 500 rows 26 cols
[1] 500 26
> head(geneData, n=2)                   # look at first two rows
                      A         B         C         D        E
```

```
AFFX-MurIL2_at  192.742  85.7533 176.7570 135.5750 64.4939
AFFX-MurIL10_at  97.137 126.1960  77.9216  93.3713 24.3986
                      F        G       H       I        J
AFFX-MurIL2_at  76.3569 160.5050 65.9631 56.9039 135.6080
AFFX-MurIL10_at 85.5088  98.9086 81.6932 97.8015  90.4838
                      K       L       M       N       O
AFFX-MurIL2_at  63.4432 78.2126 83.0943 89.3372 91.0615
AFFX-MurIL10_at 70.5733 94.5418 75.3455 68.5827 87.4050
                      P        Q       R       S       T
AFFX-MurIL2_at  95.9377 179.8450 152.467 180.834 85.4146
AFFX-MurIL10_at 84.4581  87.6806 108.032 134.263 91.4031
                        U        V       W        X       Y
AFFX-MurIL2_at  157.98900 146.8000 93.8829 103.8550 64.4340
AFFX-MurIL10_at  -8.68811  85.0212 79.2998  71.6552 64.2369
                       Z
AFFX-MurIL2_at  175.6150
AFFX-MurIL10_at  78.7068
```

These are Affymetrix microarray data, with each row, representing expression data of a gene for 26 samples. Altogether, there are 500 genes hence 500 rows. For more information:

```
> help(geneData,package = ''Biobase'')
```

What we want to do with this data is to calculate the correlation coefficients of gene expression values between all pairs of genes for these 26 samples, in order to find out which genes are under shared regulation. To calculate the correlation between two genes, say gene 12 and 13:

```
> cor(geneData[12, ], geneData[13, ])
[1] 0.548477
```

This is a small data set comparing to real life data, but even with the 500 genes, the number of gene pairs (or correlation tests) is still considerable: 500x499/2 = 124,750. Let's generate the gene pairs:

```
> pair <- combn(1:nrow(geneData), 2, simplify = F)
> length(pair)
[1] 124750
> head(pair, n = 3)
[[1]]
[1] 1 2
[[2]]
[1] 1 3
[[3]]
[1] 1 4
```

```
> tail(pair, n = 3)
[[1]]
[1] 498 499
[[2]]
[1] 498 500
[[3]]
[1] 499 500
```

To calculate correlation for all gene pairs, we will use the lapply function, which takes a list as its first argument, and pass each element from the list to a function specified as the second argument. The return value from lapply is a list with the same length with the input list, and each output element corresponds the input element processed by the user-specified function.

We will first define our own function that accepts one element from the variable "pair" and returns the the correlation coefficient between the pair of genes specified in the input element:

```
> geneCor <- function(x, gene = geneData) {
+      cor(gene[x[1], ], gene[x[2], ])
+ }
```

If we want to calculate the correlation between gene 12 and gene 13:

```
> geneCor(c(12, 13))
[1] 0.548477
```

Now we want to calculate correlation coefficients for the first 3 pairs of genes using lapply and our geneCor:

```
> out <- lapply(pair[1:3], geneCor)
> out
[[1]]
[1] 0.1536158
[[2]]
[1] 0.7066034
[[3]]
[1] -0.2125437
```

To calculate the time taken to do the correlation between all pairs stored in our variable "pair":

```
> system.time(out <- lapply(pair, geneCor))
user system elapsed
14.440 0.000 14.444
```

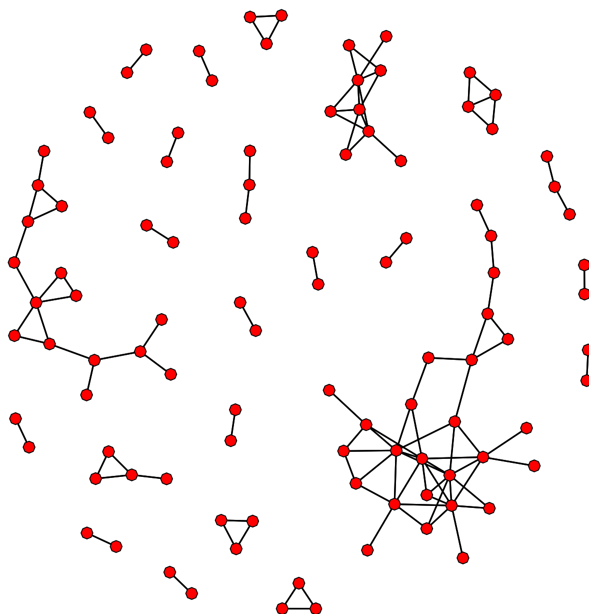
It spent 14 seconds or so (depending on how powerful the machine you are using) for all 124,750 pairwise correlation calculations. So looks like it is not really that

necessary to do parallel computing since it is so fast (especially when considering the parallelization overhead). However, many microarray calculations involve far more than 26 samples and more than 500 genes, and also, we will need to do more than just a simple correlation calculation. For our tutorial, this will be our starting example of parallel computing in R.

To visualise the output, we can draw a network of the genes above correlation threshold of 0.86 or below -0.86 (for negative correlation):

```
> corm <- cbind(do.call(rbind, pair), unlist(out))
> corm <- corm[abs(corm[,3]) >= 0.86, ]      # remove low cor pairs
> library(network); library(sna)
> net <- network(corm, directed = F)         # the network
> cd <- component.dist(net)                  # component analysis
> delete.vertices(net, which(cd$csize[cd$membership] == 1))
                              # delete genes not connected with others
> plot(net)
```

The above code produces the network figure:



The job was running on a Quad-core computer, however, only one core was used, as you can see from the CPU meters, of the four computing cores, only one is used 100% while the others are hardly touched:

```
1 [                                                 0.0%]
2 [################################################ 100.0%]
3 [*                                                0.7%]
4 [                                                 0.0%]
```

Since each pairwise comparison is independent of each other, why not utilising all four cores to do the computation? How about distributing each pairwise comparison to 5 machines with 4 cores each?

This is the basic scenario for this tutorial. In the following sections, we will apply parallel computing approaches to speed up our tasks and analyse how it is done and where the speedups and the bottlenecks are.

2.2 Multicore Computers

If you bought a new computer recently, I bet it has a multi-core processor. However, as shown above, R processes will only use a single core. How can we utilize all cores for our calculation?

It turns out to be surprisingly easy, thanks to the multicore package by Simon Urbanek (http://cran.r-project.org/web/packages/multicore/index.html). If you have not installed this package yet, it is now time to type:

```
> install.packages("multicore")    # in case you haven't installed it
```

The simplest use of the multicore package is to substitute the lapply function with mclapply:

```
> library(multicore)
> system.time(out <- mclapply(pair, geneCor))
   user system elapsed
 14.960 14.890 3.892
```

Great! By simply adding two letters in our original code (*mc*lapply), we get 3.7 times speed up (3.892 seconds instead of 14 seconds) on a quad-core computer, where all four cores are fully utilized:

```
1 [################################################ 100.0%]
2 [################################################ 100.0%]
3 [################################################ 100.0%]
4 [################################################ 100.0%]
```

2.3 Simple Network Of Workstations (snow)

Now suppose we have a much larger data set (which we will invent simply by taking our familiar 500x26 dataset and catenating them into a four times larger set as follows using the cbind or column binding function):

```
> fakeData <- cbind(geneData, geneData, geneData, geneData)
```

This means that we now have a matrix of 500 x (26 x 4), for us to perform 124750 pairs but this time each pair is involves correlating not 26 but 104 pairs of data.

To make things complicated, you are no longer satistified with simple correlation coefficients. Instead, you want to find the 95% confidence intervals of the correlation coefficients, through bootstrapping. So you define a new version of geneCor:

```
> library(boot)
> geneCor2 <- function(x, gene = fakeData) {
+     mydata <- cbind(gene[x[1], ], gene[x[2], ])
+     mycor <- function(x, i) cor(x[i,1], x[i,2])
+     boot.out <- boot(mydata, mycor, 1000)
+     boot.ci(boot.out, type = ‘‘bca’’)$bca[4:5]
+ }
> geneCor2(c(12, 13))
[1] 0.3999350 0.6580364
```

Let’s try 10 pairs of genes first:

```
> system.time(out <- lapply(pair[1:10], geneCor2))
   user  system elapsed
  2.270   0.010   2.281
```

Oops, It is too slow, we need 8 hours for all pairs of genes. For the purpose of this tutorial, we should calculate a small subset (0.24%) of gene pairs from now on:

```
> pair2 <- sample(pair, 300)
> system.time(out <- lapply(pair2, geneCor2))
   user  system elapsed
 63.230   0.310  63.545
> system.time(out <- mclapply(pair2, geneCor2))
   user  system elapsed
 66.250  66.250  19.085
```

The multicore package improved the speed by 3.3 times, however your boss thinks you should do better, because you have SSH access[1] to four Linux computers in your lab. Unfortunately for you, the computers are not connected as a cluster due to various reasons. How can you fully utilize all those computers without directly handling each individual one?

Thanks to Luke Tierney and his colleagues, you can build a Simple Network of Workstations (snow) on the fly, through the snow package (http://cran.r-project.org/web/packages/snow/index.html). Again, if you haven’t installed the snow package yet, it is time to do this and connect up say four machines, namely, chaos, variome, lams and bug:

[1] If you don’t wish to type your password every time you call all four machines, you can make setup public key authentication (check out the command ssh-keygen and ssh-copy-id).

```
> library(snow)
> hosts <- c("chaos", "variome", "lams", "bug")
> cl <- makeCluster(hosts, type = "SOCK")
```

The first question we wish to answer is this: Can we send a simple command to several machines and have them run the command simultaneously? Here's how we can call each machine to print the current date and time.

```
> clusterCall(cl, date)
[[1]]
[1] "Tue Apr 13 19:24:59 2010"
[[2]]
[1] "Tue Apr 13 19:23:36 2010"
[[3]]
[1] "Tue Apr 13 19:37:42 2010"
[[4]]
[1] "Tue Apr 13 19:24:10 2010"
```

Oops, these machines don't look as if their system times are totally synchronised! Let's get them each to print their node name and the type of machine as follows:

```
> clusterCall(cl, function(x) Sys.info()[c("nodename", "machine")])
[[1]]
nodename  machine
"chaos" "x86_64"
[[2]]
nodename   machine
"variome"    "i686"
[[3]]
nodename  machine
  "lams"   "i686"
[[4]]
nodename  machine
   "bug"   "i686"
> stopCluster(cl)
```

So as you can see, it is perfectly possible from one machine to send commands to all four machines simultaneously and get them all to run the same command. Note that you should stop the cluster every time you don't need to use them, by using the stopCluster() command.

Let's get back to work. So you have 4 computers — one quad-core, two dual-core, and one old single-core. Both R and the snow package of course, have to be pre-installed on the four machines for this to work. Now we can parallelize our work as follows:

```
> library(snow)
> hosts <- c(
+    "chaos", "chaos", "chaos", "chaos",        # the quad core
+    "variome", "variome",                      # the first dual core
+    "lams", "lams",                            # the second dual core
+    "bug"                                      # the single core
+ )
> cl <- makeCluster(hosts, type = "SOCK")
> clusterExport(cl, "fakeData")
> clusterEvalQ(cl, library(boot))
> system.time(out <- clusterApplyLB(cl, pair2, geneCor2))
   user  system elapsed
   0.12    0.00   10.92
```

OK, through using a SNOW network of four multicore-machines, we have achieved 1.75 times speed up compared to the multicore computing, or 5.8 times improvement comparing to single-core computing. The speed improvement from multicore is not spectacular, and there are at least two important reasons for that — firstly, the newly added computers are not as powerful as the original one; secondly, the communication overhead between computers is much heavier than within one computer. Therefore, increase in the number of machines or cores, does not automatically mean a great speedup because we have to account for the overheads. So one has to be careful to make your calculations whenever you are thinking about scaling up your computation over a cluster (or a grid or a cloud for that matter).

As you should have noticed, we need to export the fakeData to all nodes in the snow cluster, but that was not required for the multicore version. This is another huge advantage for the multicore package — all parallel jobs share the full state of R process when spawned, and therefore there is no data or code copying and initialization required, which leads to more efficient parallel computing. This is the third important reason that you need to be mindful of whenever you are thinking about computational speedups, distributed computing and simple parallelisation.

2.4 Linux Cluster

The snow package supports a number of parallel computing back-ends, such as MPI, which makes implementing parallel computing on existing computer clusters very easy. For example, our university provides us a Linux cluster (http://www.nus.edu.sg/comcen/HPC/), where we can submit jobs to run on up to 32 processors. (Detailed instructions on the usage of the HPC cluster can be found here: https://inetapps.nus.edu.sg/comcen/hpc/supportnews/techinfo.html).

```
# submit our script through Open MPI environment
#
# single processor
> system.time(out1 <- lapply(pair2, geneCor2))
```

```
    user  system elapsed
 70.249   0.116  70.377
#
# 1 master + 31 worker processors
> library(snow)
> cl <- makeCluster()
> clusterExport(cl, "fakeData")
> clusterEvalQ(cl, library(boot))
> system.time(out2 <- clusterApplyLB(cl, pair2, geneCor2))
    user  system elapsed
  2.531   0.000   2.531
> stopCluster(cl)
```

So, we have achieved 27.8 times speed up by using one master and 31 worker processors. This computation now takes two seconds (not considering the job submitting and queuing time) — almost as good as instantaneous!

2.5 *"One Ring to Rule Them All"*

OK. You don't want to maintain three versions of your code for running on these different systems in various parallel or non-parallel environments. Here is the solution.

2.5.1 The foreach Framework

The foreach and iterators packages created by REvolution Computing provide us a convienient framework for parallel computing in R. With these two packages, you only need to write one version of your code for all parallel backends.

```
> require(foreach)
> out <- foreach(p = pair2) %dopar% { geneCor2(p) }
```

This code will run in all parallel and non-parallel environments. To handle different backends, what you need to do is to simply register the backend with foreach (see next part). The iter function from the iterators package will create an iterator for the list pair2. Then the %dopar% operator will execute the statements that follows %dopar%.

The framework is very flexible. You can iterate through various data sources — vectors, lists, matrices, data frames, customized function, database entries, etc. You can create your own iterators too. If communication overhead is heavier than your actual computing, such as the simple correlation coefficient in the beginning of this tutorial, you can consider chunking the tasks through iterators. More details about the iterators package can be found at http://cran.r-project.org/web/packages/iterators/index.html.

The foreach function is built upon iterators. It accepts several options. For example, you can specify how to combine the %dopar% results using the .combine option. Multiple foreach-es can be nested using %:%. More details about the foreach

package can be found at http://cran.r-project.org/web/packages/foreach/index.html.

With the flexibility, the foreach and iterators packages themselves are valuable tools in R. However, their real power only comes when you register a parallel backend with them.

2.5.2 Register a Parallel Backend

Several packages were created by REvolution Computing to enable different parallel backends for the foreach code:

- doMC: multicore computer, using the multicore package
- doSNOW: snow clusters or any clusters supported by the snow package
- doMPI: clusters supporting MPI, using the Rmpi package

To register the backends with foreach is very easy, for example to register a multicore backend:

```
> registerDoMC()
```

Or, to register a MPI cluster:

```
> cl <- startMPIcluster()
> registerDoMPI(cl)
```

Simply registering a parallel backend to your foreach code, you are good to go with parallel computing in R. (If you don't register a parallel backend, your code will run just fine in serial mode)

2.5.3 Serial Code

Let's first solve our example in sequential manner, using the foreach framework:

```
#!/usr/bin/env Rscript
library(foreach)
data(geneData, package = 'Biobase')
pair <- combn(1:nrow(geneData), 2, simplify = F)
fakeData <- cbind(geneData, geneData, geneData, geneData)
pair2 <- sample(pair, 300)
print(system.time(
    out <- foreach(p = pair2, .packages = 'boot', .combine = 'rbind') %dopar%
    {
        mydata <- cbind(fakeData[p[1],], fakeData[p[2], ])
        mycor <- function(x, i) cor(x[i,1], x[i,2])
        boot.out <- boot(mydata, mycor, 1000)
        ci <- boot.ci(boot.out, type = 'bca')$bca[4:5]
```

```
        c(p, ci)
    }
))
print(head(out))  # print the head of the result
```

This script is named as base.r, and it can itself work as normal R code:

```
$ ./base.r
   user  system elapsed
 65.920   0.440  66.355
         [,1] [,2]       [,3]       [,4]
result.1  288  422 -0.3357138 -0.1266808
result.2  243  303 -0.1529784  0.1017721
result.3  234  386 -0.1561772  0.2161708
result.4    5  183  0.2505966  0.4794503
result.5  275  440  0.2420215  0.5738407
result.6   62  457  0.1558987  0.4712927
```

The above code is the ONLY version that we need to write to solve our problem. To enable parallel computing for this piece of code, we simply register various parallel back-ends with it.

2.5.4 doMC

Let's start with our multicore machine:

```
#!/usr/bin/env Rscript
library(doMC)
registerDoMC()
source("base.r")     # or paste the shared code here
```

That's it! Except the parallel backend register code, we do not need do any modification to our original code and we get a multicore parallel code!

```
$ ./doMC.r
   user  system elapsed
 56.410  56.290  19.114
         [,1] [,2]       [,3]        [,4]
result.1  164  172 -0.2668234  0.14122971
result.2   31   67  0.6607894  0.78171239
result.3   62  282 -0.3225335 -0.03370322
result.4   37  293  0.1073143  0.31629694
result.5   38  271 -0.1041704  0.28953027
result.6  211  287 -0.3495002  0.00608487
```

2.5.5 doSNOW

Let's port our code to a SNOW netowrk:

```
#!/usr/bin/env Rscript
library(doSNOW)
hosts <- c(
   'localhost', 'localhost', 'localhost', 'localhost',
   'variome', 'variome',
   'lams', 'lams',
   'bug'
)
cl <- makeCluster(hosts, type = 'SOCK')
registerDoSNOW(cl)
source("base.r")    # or paste the shared code here
stopCluster(cl)
```

So, the only things we did here are:

1. choose computing hosts
2. start and register our SNOW cluster
3. run our original code!
4. stop the cluster

Let's test this code out:

```
$ ./doSNOW.r
   user  system elapsed
  0.130   0.020  12.335
         [,1] [,2]       [,3]        [,4]
result.1  217  332  0.1224703  0.44764756
result.2   19  179 -0.6041616 -0.36109986
result.3  192  426  0.4824728  0.68836742
result.4   46  228 -0.2883579 -0.01718258
result.5  407  426  0.1998271  0.45557312
result.6  165  386 -0.6302590 -0.15638907
```

2.5.6 doMPI

How about MPI cluster code?

```
#!/usr/bin/env Rscript
library(doMPI)
cl <- startMPIcluster()
registerDoMPI(cl)
source("base.r")   # or paste the shared code here
closeCluster(cl)
```

Great, it is even simpler than the SNOW version.

```
$ bsub -o out -a openmpi -n 32 mpirun.lsf ./doMPI.r
$ (wait, and then) cat out
   user  system elapsed
  2.742   0.243   3.015
         [,1] [,2]         [,3]       [,4]
result.1  202  231  0.003461775  0.3608233
result.2  174  372  0.009071919  0.3583579
result.3  118  186  0.601903906  0.7885814
result.4  186  362 -0.336281370 -0.1353606
result.5  270  485 -0.491912888 -0.1121098
result.6  168  413 -0.147680243  0.1079481
```

Three seconds! From the above examples, I hope you can see the power of unified parallel computing framework by foreach.

3 Parallel R in Cloud

There are several ways to do parallel R computing in Cloud. The simplest is to boot up a multi-core instance and use the multicore package (Section 2.2). If a single machine is not powerful enough for you, you can boot up multiple instances and link them as a snow cluster (Section 2.3) or a Linux cluster supporting MPI (Section 2.4). In all cases, you can use the foreach framework to write a single version of your code, which is able to run on all the platforms (Section 2.5).

However, we are not using the full power of Cloud. For example, if you have massive amount of data to analyze, you should try the RHIPE package, which is an *R and Hadoop Integrated Processing Environment*. It uses a mapReduce framework to handle massive amount of data in Cloud. More details about the RHIPE package can be found at http://www.stat.purdue.edu/~sguha/rhipe/.

4 Useful Resources

- http://cran.r-project.org/web/views/HighPerformanceComputing.html
- http://cran.r-project.org/manuals.html