Parallel Computing with R using SPRINT on post-genomic data

Swansea, 29-30 May 2014

Workbook

SPRINT Examples

Contents

1. Introduction

The motivation for the SPRINT project is a) accessibility of High Performance Computing for regular R users without parallelisation expertise and b) avoiding redundancy by making available already parallelised function solutions.

SPRINT (Simple Parallel R INTerface) is a parallel framework for R. It provides a High Performance Computing (HPC) harness that allows R scripts to run on HPC clusters.

SPRINT contains a library of selected R functions that have been parallelized. Functions are named after the original R function with the added prefix 'p', i.e. the parallel version of *cor()* in SPRINT is called ***pcor()***. Calls to the parallel R functions are included directly in standard R scripts.

These examples show you how to edit R scripts to use SPRINT, and how to submit these to run in parallel on the HPC Wales platform.

!!! Important !!!

Where a line in this workbook is preceded by "$", this means you will be typing this on the command line of your operating system with a terminal program.

2. Exercise set up

2a. Start up your terminal application/program on your laptop/device

Interaction with the HPC (and most other HPC systems) is through the 'SSH' protocol, which is used via command line or graphical user interface in some cases.

-> On a Mac OS, this is 'Terminal' (or 'iTerm' or any other you have downloaded yourself)

-> On a Unix-like OS, this is the terminal window.

-> On a Windows computer, this is PuTTY and you need to download it yourself you're your laptop(www.putty.org). The lab computers already have PuTTY installed.

2b (Windows). Login to the HPC Wales network using PuTTY on Windows:

Start the PuTTY application.

```
Configuration:
host name: login.hpcwales.co.uk
select ssh (probably the default option).

click 'open' button (If you get a message re "authenticity of
host […] can't be established", just enter  'yes' to continue")

login as: enter your username here
password: enter your password here.
```

2b (Mac or Unix). Log in to the HPC Wales Network using the terminal window:

```
From your terminal window, log in to the HPC Wales Network
$ ssh username@login.hpcwales.co.uk
```

If you get a message re "authenticity of host […] can't be established", just enter 'yes' to continue".

2b. From the frontend of HPC Wales, log in to the cluster at Aberystwyth and set up the file structure.

(This creates a working directory for these exercises)

```
1. Once on the HPC Wales Network, login to one of the available
compute clusters (for this course 'ab-log-001')
$ ssh ab-log-001

2. Create a directory/folder for this workbook, and go into it
$ mkdir sprint
$ cd sprint
```

2c. Copy our exercises to your 'sprint' folder:

```
    1. Load up the http proxy module to get access to the internet,
then download the exercises.tar file.

    $ module load http-proxy
    $ wget http://www.archer.ac.uk/training/course-
    material/2014/05/SPRINT_HPCWales/Exercises/exercises.tar
```

2. Uncompress ('unzip') the file and go into the newly unpacked directory called 'exercises'.

```
$ tar -xvf exercises.tar
$ cd exercises
```

3. HPC Wales reservations

For the two days of this course there is reserved time on the HPC Wales machine at Aberystwyth. Your examples will run faster if you submit them using these reservations as described below. The reservation name for the course is 'rsprint':

```
$ export RES="rsprint"
```

Important! You will have to re-enter this command every time you log in.

4. SPRINT installation test

First, we'll run a very small R script that calls the SPRINT ptest() method. The ptest() method just prints hello from each processor, and is useful for checking that you are running SPRINT in parallel. We need 2 files for this, their contents are shown below.

1) An R script containing the necessary code to load the SPRINT library, execute the testing function and quit SPRINT. This is file "`sprint_test.R`"

2) HPC Wales shell script that starts R, tells R where the used directories are, names the R script to be run and finally sets up the HPC Wales command line options for how to run the named R script in parallel. This is file "`sub.q`".

sprint_test.R

```
library("sprint")
ptest()
pterminate()
quit()
```

sub.q

```
#!/bin/bash --login
# ! Edit number of processors to fit your job
#BSUB -n 8
# ! Redirect stdout to the file filename
#BSUB -o sprint_test.o.%J
# ! Redirect sterr to the file filename
#BSUB -e sprint_test.e.%J
# ! Edit the job name to identify separate job
#BSUB -J sprint_test
# ! Edit time to fit your job
#BSUB -W 0:10

module purge
module load SPRINT

mpirun -n 8 R -f sprint_test.R
```

You don't need to edit either of the files.

4a. Submit an executable file ("sub.q") to the HPC Wales queue with the 'bsub' command

```
$ bsub -U $RES < sub.q
```

You should then see:

```
Job <xxxxxx> is submitted to queue <q_ab_mpc_work>.
```

4b. Find out if the job is still waiting on the queue or has finished:

```
$ qstat -u $USER
```

If it says 'No matching job found', then the job has finished running and there should be two new files in your directory called `sprint_test.e{job number}` and `sprint_test.o{job_number}`.

4c. Have a look at `sprint_test.o{job_number}`. You can type 'less sprint_test.o', then hit the tab key to autocomplete.

```
$ less sprint_test.o{job_number}
```

Near the end of the file (press the space bar to move downwards in the screen output, press 'q' key to exit the 'less' program) you should see

```
> ptest()
[1] "HELLO, FROM PROCESSOR: 0" "HELLO, FROM PROCESSOR: 2"
[3] "HELLO, FROM PROCESSOR: 3" "HELLO, FROM PROCESSOR: 4"
[5] "HELLO, FROM PROCESSOR: 5" "HELLO, FROM PROCESSOR: 6"
[7] "HELLO, FROM PROCESSOR: 7" "HELLO, FROM PROCESSOR: 1"
```

in the output. If not, something has gone wrong. Read the `sprint_test.e{job_number}` file to see the error message.

```
$ less sprint_test.e{job_number}
```

5. Introduction to Exercises

Each of the following exercises introduces a different SPRINT function, but they all follow the same structure.

You will be given an R script made up of standard R commands and two .q files for submitting the R files to HPC Wales.

You will run the R script using the serial .q file and note the time taken to run.

Next make a copy of the R script and edit it to use SPRINT.

Use the parallel .q file to run it and note the run time. You can compare this to the time obtained in the serial run.

6. Exercise 1: pcor () and ppam()

This exercise uses microarray gene expression data and identifies similarity in gene expression profiles across all biological conditions and subsequently uses this information to cluster (group) genes by this similarity. The data set used is Golub et al, a microarray study of patients with different types of leukemia, and commonly used to compare the performance of old or new machine learning algorithms (clustering or classification of genes or samples).

Of course, measuring of pairwise distances and clustering of such data is not limited to gene expression data, any numerical data sets will do.

The first step in this exercise measures similarity between all possible pairs of genes, using simple Pearson correlation (cor(); we are working on other distance metrics). The resulting correlation matrix (aka similarity, adjacency or guilt-by-association matrix) is then used as input for the Partitioning-Around-Medoids clustering algorithm (pam()), which identifies distinct sets of genes that are more similar to one another than to genes in other such groups. This explorative analysis usually results in “interesting” gene expression patterns, where the assumption is that gene co-expression across samples may indicate biological co-regulation.

The output of these functions are usually a numeric correlation matrix of large size, and for PAM a data object containing information allowing the plotting of gene expression profiles in their various clusters.

For this exercise, memory (RAM) rather than CPU speed is the frequently limiting factor. While this data set is relatively small (to keep the exercise short), the computation of large correlation matrices (square of the number of rows in the input data set) is challenging for larger data sets and quickly exceeds available memory. SPRINT both speeds up the computation time and allows the generation of larger correlation matrices by using harddrive space as extra RAM if necessary [1][2].

Exercise 1 Serial

This section introduces the R code necessary to run pairwise gene-gene correlations and PAM clustering without any parallelization.

6a. First have a look at the R script using the serial version of cor() and pam(). See also *Appendix 1: Code for exercises*. In essence, in this R script the Pearson correlation matrix between all genes is computed and then used as the distance metric in a PAM clustering. We’ve added some code that outputs the system time taken for these computations.

```
$ less exercise1_serial.R
```

6b. Submit the job to HPC Wales:

```
$ bsub -U $RES < sub_ex1_serial.q
```

6c. Check the progress, then read the output file and note the timings in the table below:

```
$ qstat -u $USER
$ less ex1_serial.o{job_number}
```

Exercise 1 Comparison of Serial and Parallel

Note the correlation time and clustering time below:

	Correlation time	**Clustering time**
serial		
parallel		

Exercise 1 Parallel

This section makes the necessary changes to the regular (serial) R script in order to instead use parallelised versions contained in SPRINT.

6d. Make a copy of the serial version of the R script:

```
$ cp exercise1_serial.R exercise1_parallel.R
```

6e. Edit (here using EMACS text editor) the new R file to use parallelised SPRINT functions instead

(use Control-x Control-s to save in emacs; Control-x Control-c to exit with saving; i.e. hold down Control key while pressing first ‘x’ key then ‘s’ or ‘c’ key):

```
$ emacs exercise1_parallel.R
```

Changes to be made the R code to use SPRINT:

Load the SPRINT library at the top of the R script (just remove comment-#):

```
library(sprint)
```

Use pcor instead of cor:

```
distance_matrix <- pcor(t(data))
```

Use ppam instead of pam.

```
# pam_result <- pam(distance_matrix, k=6, diss=TRUE)
pam_result <- ppam(distance_matrix, k=6)
```

Note 1: diss is automatically detected to be true (recognising the input to be a (dis)similarity matrix rather than the source microarray data), and so the diss parameter is not passed to the ppam() method.

Note 2: In a real application it is more appropriate to provide PAM with a dissimilarity (distance) rather than a similarity (correlation) matrix. This would here more simply be achieved by converting cor() to 1-cor().

Terminate SPRINT in your R code once you have finished using it:

```
pterminate()
```

Save the file (use Ctr-x Ctr-s to save in emacs, or Ctr-x Ctr-c to exit and confirm save).

6d. Submit the parallelised SPRINT job to HPC Wales

The sub_ex1_parallel.q HPC Wales shell script (provided in your work directory) is almost identical to the shell script submitting the serial R script . The main difference between the serial and parallel submission scripts is that the serial script calls R directly and the parallel script calls mpirun to run on 8 processes, looking like this:

```
mpirun -n 8 R --no-save --quiet -f exercise1_parallel.R
```

Submit the job:

```
$ bsub -U $RES < sub_ex1_parallel.q
```

Check the progress, then read the output file and note the timings in the table above:

```
$ qstat -u $USER
$ less ex1_parallel.o{job_number}
```

N.B. The Max Processes and Max Threads reported are taken as a snapshot in time, and as this code runs very quickly, the actual numbers of threads and processes may be under-reported.

7. Exercise 2: pRP()

This exercise uses the CLL package, which contains chronic lymphocytic leukemia (CLL) gene expression data. There are 12,625 genes and 22 samples that were either classified as progressive or stable.

This exercise differs from the previous one in that the focus of a Rank Product analysis is statistical inference for each gene. Its purpose of identifying (for a given gene) significant expression changes between two biological conditions is identical to that of a t-test, but it works on a different principle. Briefly, Rank Product analysis uses expression ratios between all pairs of samples from two biological groups as the primary statistic, ranks these in relation to all other genes, and the rank product of each gene is finally compared to a Null distribution of the same statistic based on permuted input data.

Output of this function is an object containing, for each gene, statistical significance of the observed differential expression between two conditions.

This analysis is computationally expensive with respect to CPU time (not RAM) due to the row permutations of the input data set. Although this problem can be limited by using small data sets (as used here) or specifying only small numbers of permutations, this is a limiting issue for larger data sets (exon arrays, possibly quantitative results from high-throughput sequencing, and other research disciplines than biology)[3][4].

Exercise 2 Serial

7a. First have a look at the serial version of the R script for running a Rank Product analysis RP(). See also *Appendix 1: Code for exercises*

```
$ less exercise2_serial.R
```

7b. Then submit the job to HPC Wales:

```
$ bsub -U $RES < sub_ex2_serial.q
```

Check the progress (this may take a little longer than exercise 1), then read the output file and note the timings in the table below:

```
$ qstat -u $USER
$ less ex2_serial.o{job_number}
```

Exercise 2 Comparison of Serial and Parallel

Note the rank product execution time below:

	rank product execution time
serial	
parallel	

Exercise 2 Parallel

Make a copy of the serial version of the code:

```
$ cp exercise2_serial.R exercise2_parallel.R
```

Edit the R file to use SPRINT

Then edit the copy to use SPRINT (use Ctr-x Ctr-s to save in emacs):

```
$ emacs exercise2_parallel.R
```

Changes to be made the R code to use SPRINT:

Load the SPRINT library:

```
library(sprint)
```

Use pRP instead of RP:

```
rp <- pRP(data, cl=classes, num.perm=50, logged=FALSE)
```

Terminate SPRINT once you have finished using it:

```
pterminate()
```

Submit the SPRINT job

Submit the job:

```
$ bsub -U $RES < sub_ex2_parallel.q
```

Check the progress, then read the output file and note the timings in the table above:

```
$ qstat -u $USER
$ less ex2_parallel.o{job_number}
```

8. Exercise 3: papply()

Function apply() is a do-whatever-you-want function but used here to apply a user-defined function to each individual array in the Golub microarray data set. The function specified here computes 1000 resampled means for each array and returns the estimated variation of this mean for each array. This is a type of bootstrap estimate and can be computationally expensive with respect to CPU time. Note that the function used here is just a simple worked example, a more sensible (but less visually accessible) bootstrap would be for example to obtain a bootstrapped expression fold change estimate between the two leukemia groups for each gene.

Computing a single statistic or metric on each row or column in a data set is normally straightforward to parallelise ("task farm"), and other packages exist that do this. It is included in SPRINT because it may be a common part of workflows around other SPRINT functions [5].

If the statistic or metric is complex or applied to a large data set, CPU time is a limiting factor for serial computation, with great benefits in parallelization.

Exercise 3 Serial

First have a look at the serial version of this code which uses apply(). See also *Appendix 1: Code for exercises*

```
$ less exercise3_serial.R
```

Then submit the job:

```
$ bsub -U $RES < sub_ex3_serial.q
```

Check the progress, then read the output file and note the timing in the table below:

```
$ qstat -u $USER
$ less ex3_serial.o{job_number}
```

Exercise 3 Comparison of Serial and Parallel

Note the apply time below:

	apply time
serial	
parallel	

Exercise 3 Parallel

Make a copy of the serial version of the code, then (as in the previous 2 exercises) edit the parallel script and submit this to the HPC Wales queue, then record execution time from the output in the above table.:

```
$ cp exercise3_serial.R exercise3_parallel.R
```

Edit the R file to use SPRINT

Use papply instead of apply. Note that that because the papply() function combines the functionality of apply() and lapply(), it has a slightly different interface than the apply function, as shown:

```
#result <- apply(data, 2, my_stat)
result <- papply(data, my_stat, 2)
```

or

```
#result <- apply(X=golub_data, FUN=my_stat, MARGIN=2)
result <- papply(data=golub_data, fun=my_stat, margin=2)
```

Remember to terminate SPRINT once you have finished using it.

Edit the submission script to run R in parallel

Make a copy of the serial submission file and edit it run the ‘exercise3_parallel.R’ code.

```
$ cp sub_ex3_serial.q sub_ex3_parallel.q
$ emacs sub_ex3_parallel.q
```

You will need to:

- Change the number of processors requested from 1 to 8.
- Edit the output filenames and job name
- Edit the last line to run R using mpirun on 8 processors.

As an example, you can see the exercise 2 submission script by running:

```
$ more sub_ex2_parallel.q
```

Submit the SPRINT job

Submit the job:

```
$ bsub -U $RES < sub_ex3_parallel.q
```

Check the progress, then read the output file and note the timings in the table above:

```
$ qstat -u $USER
$ less ex3_parallel.o{job_number}
```

9. Interactive mode for debugging

The batch submission process described above is the standard way to interact with the HPC Wales system. However, it can be useful to run your code interactively for debugging purposes only. This method should not be used for large amounts of processing. By default, you will not be able to run an MPI job interactively on the login node.

Load the interactive MPI environment and SPRINT as before.

```
$ module purge
$ module load SPRINT
$ I_MPI_FABRICS=shm
```

Run R interactively on one process (most of the SPRINT methods will work, apart from pcor()).

```
$ R
> library(sprint)
… Your own R code here.
```

Run the code on several processes; the output will appear directly in you window:

```
$ mpirun -n 8 R --no-save -f your_code_file_here.R
```

10. Appendix 1: Code for exercises

Exercise 1

```
# SPRINT: Parallel computing with R on HPC Wales
# 13-May-14
# Exercise 1: pcor() and ppam()

library(cluster)
# Combined Test and Training Sets from the Golub Paper
library(golubEsets)

#===============================================================================
#                         Load Data
#===============================================================================


# The data are from Golub et al. These are the combined training samples and
# test samples. There are 47 patients with acute lymphoblastic leukemia (ALL)
# and 25 patients with acute myeloid leukemia (AML). The samples were assayed
# using Affymetrix Hgu6800 chips and data on the expression of 7129 genes
# (Affymetrix probes) are available.

data(Golub_Merge)
data <- exprs(Golub_Merge)

#===============================================================================
#                         SPRINT
#===============================================================================
#library(sprint)

stime <- proc.time()["elapsed"]
# Calculate distance matrix using correlation function. The parallel version
writes
# its output to a file that is loaded as an ff object in R and behaves (almost)
# as if data was stored in RAM
distance_matrix <- cor(t(data))

etime <- proc.time()["elapsed"]
print(paste("Correlation time: ")); print(paste(etime-stime))

# Force memory clean-up
gc(reset=TRUE, verbose=FALSE)

stime <- proc.time()["elapsed"]
```

```
# Find 6 medoids using the PAM algortithm
# You are passing a distance matrix on input, so set the diss parameter to TRUE
# The parallel version of the algorithm will automatically detect the format of
# the input data so no additional parameter is required

pam_result <- pam(distance_matrix, k=6, diss=TRUE)

etime <- proc.time()["elapsed"]
print(paste("Clustering time: ")); print(paste(etime-stime))

# do further analysis of data then exit MPI
# ...
#
# You can always save your data or objects on disk and continue to work on it
# in your interactive R window

# Shutdown the SPRINT library
#pterminate()

quit(save="no")
```

Example continued on next page…

Exercise 2

```
# SPRINT: Parallel computing with R on HPC Wales
# 13-May-14
# Exercise 2: pRP()

library(RankProd)
# The CLL package contains the chronic lymphocytic leukemia (CLL) gene
# expression data
library(CLL)

#===============================================================================
#                         Load Data
#===============================================================================

# The sCLLex ExpressionSet object has 12,625 genes and 22 samples that were
# either classified as progressive or stable.
data(sCLLex)
data <- exprs(sCLLex)
cl <- phenoData(sCLLex)$Disease

# Convert factors into integers progres. -> 0 stable -> 1
classes <- unclass(cl)-1

#===============================================================================
#                         SPRINT
#===============================================================================
#library(sprint)

stime <- proc.time()["elapsed"]
# Perform rank product method
rp <- RP(data, cl=classes, num.perm=100, logged=FALSE)

etime <- proc.time()["elapsed"]
print(paste("rank product execution time: ")); print(paste(etime-stime))

# do further analysis of data then exit MPI
# ...

# Shutdown the SPRINT library
#pterminate()

quit(save="no")
```

Exercise 3

```
# SPRINT: Parallel computing with R on HPC Wales
# 13-May-14
# Exercise 3: papply()

#==============================================================================
#                 User defind function to be applied over data
#==============================================================================

my_stat <- function(x) {
      N <- length(x) # sample size
      r <- floor(N/10) #sub-sample size
      count <- 1000 # number of resamples
      res <- vector(mode = "double", length=N)

      for (i in 1:count) {
            y <- x[sample(N,N, replace=TRUE)]
            res[i] <- mean(y) }
      return(sd(res))}

# Combined Test and Training Sets from the Golub Paper
library(golubEsets)
#==============================================================================
#                         Load Data
#==============================================================================

# The data are from Golub et al. These are the combined training samples and
# test samples. There are 47 patients with acute lymphoblastic leukemia (ALL)
# and 25 patients with acute myeloid leukemia (AML). The samples were assayed
# using Affymetrix Hgu6800 chips and data on the expression of 7129 genes
# (Affymetrix probes) are available.

data(Golub_Merge)
golub_data <- exprs(Golub_Merge)
#==============================================================================
#                         SPRINT
#==============================================================================

stime <- proc.time()["elapsed"]
# We will use papply to filter the correlattion matrix
# stored in a file to save us some disk space.
# Remeber that papply has combined functionality of apply and lapply
# functions, thus it has slightly different interface than sequential
# apply function.

result <- apply(X=golub_data, FUN=my_stat, MARGIN=2)
#result <- papply(data=golub_data, fun=my_stat, margin=2)

etime <- proc.time()["elapsed"]

time <- proc.time()["elapsed"]
print(paste("Apply time: ")); print(paste(etime-stime))

# do further analysis of data then exit MPI...

# Shutdown the SPRINT library

quit(save="no")
```

11. References

1. S. Petrou SPRINTing with HECToR, 2010. http://www.hector.ac.uk/cse/distributedcse/reports/sprint/sprint.pdf.

2. M. Piotrowski, T. Forster, B. Dobrezelecki, T.M. Sloan, L. Mitchell, P. Ghazal, M. Mewsissen, S. Petrou, A. Trew, J. Hill. Optimisation and parallelisation of the partitioning around medoids function in R. International Conference on High Performance Computing and Simulation (HPCS) 2011; :707-713. doi: 10.1109/HPCSim.2011.5999896

3. R. Breitling, P. Armengaud, A. Amtmann, P. Herzyk. Rank products: a simple, yet powerful, new method to detect differentially regulated genes in replicated microarray experiments. FEBS Letters 2004; 573:83–92. DOI: 10.1016/j.febslet.2004.07.055.

4. L. Mitchell, T. M. Sloan, M. Mewissen, P. Ghazal, T. Forster, M. Piotrowski, A.Trew, 2012, Parallel classification and feature selection in microarray data using SPRINT. Concurrency and Computation: Practice and Experience, early online version, Sep 2012, http://dx.doi.org/10.1002/cpe.2928.

5. M. Piotrowski, G. A. McGilvary, T. M. Sloan, M. Mewissen, A. D. Lloyd, T. Forster, L. Mitchell, P. Ghazal, J. Hill, Exploiting Parallel R in the Cloud with SPRINT. Methods of Information in Medicine 2013 Vol. 52 (1), 80-90. http://dx.doi.org/10.3414/ME11-02-0039. PubMed PMCID: PMC3547073

SPRINT TEAM

EMAIL: SPRINT@ED.AC.UK

HTTP://WWW.R-SPRINT.ORG