# Parallelisation of bootstrap function as part of SPRINT

Laurence Alexander Baldwin

August 27, 2010

MSc in High Performance Computing The University of Edinburgh
Year of Presentation: 2010

**Abstract**

Analysing genetic data requires large amounts of processing time and memory due to the complex nature and size of genetic information. An example problem of this type is microarray based studies which can measure thousands of different genes over thousands of samples.

One of the popular tools used by biostatistions for statictic analysis of such microarray data is the R scripting language. However R is inherently serial and as such cannot take advantage of multicore machines or HPC systems.

The SPRINT project aims to make functions available for R that have been parallelised, thus allowing users to easily make use of HPC systems to improve the performance of existing R scripts.

During initial discussions with the dissertation supervisor and from feedback from the SPRINT team, the bootstrapping function was identified as an ideal function to be parallelised in this dissertation project. Bootstrapping is a method of estimating the standard error and confidence intervals of a datasets properties

Based on findings from background research SNOW and RMPI where identified as alternative methods to parallelise the bootstrapping function. Using these methods initial parallel bootstrapping implementations where developed.

The implementations in both SNOW and RMPI proved successful, with a number of implementations providing speedup. Based on these findings the project went on to implement a solution in SPRINT.

In the SPRINT implementation a number of challenges were faced. In particular the challenge of working with R objects and converting them to native C types, so that they could be sent as messages using MPI. Once these challenges were overcome a number of different implementations of parallel bootstrapping were produced using the SPRINT framework.

Benchmarks on the parallel versions of bootstrapping written using SPRINT, showed good speedup with all versions. Including a version which provided increasing speedup up to 16 processors and gave the exact same results as the serial version. This version is put forward as a candidate to be released as a beta in the SPRINT framework.

Finally the conclusions of the project are drawn and suggestions are made for possible further work that can be carried out, including suggestions to improve the ease of development in the SPRINT framework.

# Contents


**1 Introduction** **1**

**2 Background theory** **3**
2.1 The R project . . . . . 3
2.1.1 Parallel R packages . . . . . 3
2.1.2 The SPRINT framework . . . . . 4
2.2 Bootstrapping . . . . . 4
2.3 Gene analysis . . . . . 5
2.3.1 DNA Microarrays . . . . . 5
2.3.2 Statistical gene analysis . . . . . 5
2.4 The System (Ness) . . . . . 6
2.5 The Dataset . . . . . 6

**3 Parallel bootstrapping with R and parallel R packages** **8**
3.1 Analysis of R boot function . . . . . 8
3.2 Random Number Generators . . . . . 9
3.3 RMPI . . . . . 10
3.3.1 Implementation boot-RPMI1 . . . . . 11
3.3.2 Implementation boot-RPMI2 . . . . . 12
3.3.3 Implementation boot-RPMI3 . . . . . 13
3.4 SNOW . . . . . 14
3.5 Speedup of RMPI and SNOW implementations . . . . . 15

**4 Writing extensions to R in C** **17**
4.1 The R API . . . . . 17
4.2 R objects . . . . . 18
4.3 Garbage collector . . . . . 18
4.4 Calling the R evaluator from C . . . . . 19

**5 The parallel pboot( ) implementation in SPRINT** **20**
5.1 SPRINT interface to pboot . . . . . 20
5.2 Parallel strategy . . . . . 23
5.2.1 Implementation pboot1 . . . . . 23
5.2.2 Implementation pboot2 . . . . . 24
5.2.3 Implementation pboot3 . . . . . 25

5.3 Work distribution . . . 25
5.4 Communication . . . 26
5.5 Testing . . . 26
5.5.1 Component testing . . . 27
5.5.2 System testing . . . 27
5.5.3 Stability testing . . . 28

**6 Results and performance analysis** **29**
6.1 Speedup of pboot() implementations . . . 29
6.2 Scaling of pboot() implementations with different problem sizes . . . 30
6.3 Scaling of pboot( ) implementation with different number of replications 31
6.4 Scaling of pboot( ) implementation with complexity of statistical function 32
6.5 Comparison of speedup with RMPI and SNOW implementations . . . 32

**7 Conclusion** **34**
7.1 Further Work . . . 35

**A Results from benchmarks** **37**
A.1 Speedup . . . 37
A.2 Testing . . . 38
A.3 Problem Size . . . 38
A.4 Replications . . . 39
A.5 Complexity . . . 40

# List of Tables


A.1 Execution time in seconds vs number of CPUs . . . . . . . . . . . . . . . 37
A.2 Speedup vs number of CPUs . . . . . . . . . . . . . . . . . . . . . . . . 37
A.3 Standard Error result with number of replications . . . . . . . . . . . . 38
A.4 Execution time in seconds for implementation vs problem size . . . . . 38
A.5 Speedup with data size . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
A.6 Execution time in seconds for implementation vs number of replications 39
A.7 Speedup with number of replications . . . . . . . . . . . . . . . . . . . 39
A.8 Execution time in seconds for implementation vs statistical function complexity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
A.9 Speedup with complexity of statistical function . . . . . . . . . . . . . 40

# List of Figures


3.1 Analysis of boot() function from R boot library . . . . . . . . . . . . . . 8
3.2 High level parallelisation of boot . . . . . . . . . . . . . . . . . . . . . . . 11
3.3 Low level parallelisation of boot . . . . . . . . . . . . . . . . . . . . . . . 12
3.4 Optimised parallelisation of boot . . . . . . . . . . . . . . . . . . . . . . . 13
3.5 Speedup of R parallel methods with boot . . . . . . . . . . . . . . . . . . . 15

5.1 SPRINT framework structure . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.2 Standard error with various implementations . . . . . . . . . . . . . . . . 28

6.1 Speedup of pboot . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
6.2 Scaling of pboot based on problem size . . . . . . . . . . . . . . . . . . . 31
6.3 Scaling of pboot based on replication size . . . . . . . . . . . . . . . . . 31
6.4 Scaling of pboot based on problem complexity . . . . . . . . . . . . . 32
6.5 Comparison of speedup of SPRINT, RMPI and SNOW implementations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

# Acknowledgements

I would like to thank my supervisor Michal Piotrowski for his guidance and patience during this project. Along with my parents for their continual support during my studies.

# Chapter 1

# Introduction

Analysing genetic data requires large amounts of processing time and memory due to the complex nature and size of genetic information. An example problem of this type is microarray based studies which can measure thousands of different genes over thousands of samples, resulting in large datasets. The information stored in DNA is inherently complex and to be analysed statistical function have to be used.

R[1] is a software package used for general statistical analysis that can be extended with libraries. The Bioconductor [2] package includes libraries for R that provide support specifically for genomic data.

The specialised functions provided by Bioconductor and the core of R are single threaded. They do not have any support for threads or any form of message passing. This means that they are limited by the speed and memory of a single processor and are unable to make use of HPC (high performance computing) systems or multicore processors that are now commonly available.

A number of packages are available to provide messaging support to R. These packages make use of standard message passing libraries and provide an R interface to use them. It is suspected that these packages however effective are not simple enough for statisticians to take full advantage of, without first studying and understanding the principles of HPC system communications.

SPRINT is a framework being developed to allow R to make full use HPC systems [3], with particular focus on bioinformatics and ease of use for biostatistions with little or no knowledge of message passing technology. The SPRINT project is under active development with a number of functions that have already been successfully implemented. Based on research by the SPRINT project the bootstrapping function *boot* in R is an ideal candidate [4] for parallelisation and widely used in bioinformatics and by statisticians in general.

In this project bootstrapping and in particular the *boot* function in R is studied and attempts made to improve the performance of the function, specifically on HPC systems.

This will be done by looking at the available packages that provide message passing

support to R and evaluating them. Then using the SPRINT framework attempting to implement a parallel version of the function.

This dissertation is divided into 6 chapters proceeding this introduction. Chapter 2 describes the background theory in understanding the problem and identifying any solutions that maybe available already.

Chapter 3 describes in detail the role random numbers play in bootstrapping. Before a variety of different methods of parallelising the *boot* function are implemented and the results are discussed.

Chapter 4 covers the technical details of using the R API to develop C extensions to R. This is essential to be able to develop solutions in SPRINT.

Chapter 5 describes how the SPRINT framework interfaces with pboot before discussing how a number of versions of the *pboot* function were implemented in SPRINT. The chapter then goes on to look at the work distribution, communication and the software testing of the implementations.

Chapter 6 takes an in depth look at the performance of the various pboot implementations. Looking at how the number of processors, size of dataset, complexity of statistical function and the number of replications, effect the performance of each implementation.

The final chapter 7 draws conclusion from all the work undertaken and discusses further work that could be undertaken.

# Chapter 2

# Background theory

## 2.1 The R project

R is a scripting language for statistical computing and graphics. It is an open source implementation of S, a language developed by Bell Laboratories [5].

As a scripting language R is easier to learn and debug than a compiled language. R scripts are very portable and can be run on any operating system or hardware that has R installed.

R has functionality to perform a wide range of statistical calculations and can easily be extended using libraries. Libraries are grouped into packages to perform related tasks. Thousands of packages are available in various statistical fields.

### 2.1.1 Parallel R packages

A number of packages are available to provide parallel functionality to R. Typically these packages are available from the CRAN repository [6]. The packages use various methods to allow R scripts to make use of any number of nodes in a HPC system.

Spreading calculations over a number of nodes allows the performance of R scripts to be improved and larger datasets to be used.

The majority of packages act as wrappers to already existing message passing libraries such as Message Passing Interface (MPI) [7] , Parallel Virtual Machine (PVM) [8] or Sockets. Some packages add another layer, allowing R scripts to be written independently of the communication layer. This is done so that scripts written using the particular package are more portable, able to run on systems running any of the supported message passing libraries.

### 2.1.2 The SPRINT framework

The SPRINT project [9] aims to allow users of R to easily exploit HPC systems, without having to deal with the complexities of parallel programming. This is achieved by providing functions that have already been parallelised and that can be used as direct replacement for the standard sequential versions. This allows users to easily modify existing scripts, calling functions that have been parallelised, to make full use of HPC systems.

The SPRINT framework is a wrapper around parallelised functions, allowing them to be easily called from inside R. It also manages a collection of processes typically one on each node of the HPC system, allowing communication between them.

SPRINT is written in C and uses MPI for message passing between nodes. The combination of C and MPI is widely used in HPC systems offering good performance and portability. MPI is commonly supported on Massively Parallel Processing (MPP) architectures, which SPRINT is therefore able to take advantage of.

A number of versions of MPI are available and SPRINT aims to support all that meet MPI2 specifications.

## 2.2 Bootstrapping

Bootstrapping is a method of estimating the standard error and confidence intervals of a datasets properties [10]. Estimated confidence intervals give a range in which the result is expected to be in and a confidence that the result is within the range.

Standard error is calculated from the standard deviation divided by the square root of the number of elements in each sample [11]. It is used to measure the difference between the original and sampled datasets.

Bootstrapping is recommended to be used in situations where the distribution of a statistic is unknown, the sample size is not large enough for statistical inference or when only a small sample is available of a larger population [12].

If the distribution of a statistic is unknown or complicated, bootstrapping provides a method of assessing the properties of the distribution. With a sample size that is not large enough to determine statistical inference, but the distribution is known, bootstrapping can be used to account for distortions of a small sample that may not be representative of the population. In studies where only a small sample is available of a larger population, bootstrapping can be used to estimate the variance of the population.

The bootstrapping technique is performed by measuring a property on a number of generated samples. Each sample is created so that it might have been seen in the original dataset.

A number of techniques can be used to generate the samples, depending on the nature of the dataset. If an element may appear a number of times in the same sample then random resampling with replacement can be used to generate the samples. Alternatively, the sample can be generated to match a specific distribution.

Bootstrapping was first introduced by B Efron in 1977, when computers started to become readily available for statistical analysis. At the time Efron stated

> "Bootstrap-like procedures have undergone very little theoretical development since they have been computationally practical for a comparatively short time, but theoreticians can be expected to take greater interest in them now that they are feasible." [13]

Over the last 30 years computer systems have developed rapidly and improved in performance exponentially. This improvement has allowed bootstrapping to be more widely used and on larger samples of data.

## 2.3 Gene analysis

### 2.3.1 DNA Microarrays

Gene microarrays are used to show the expression level of genes in a sample. The expression level being an indication of the abundance of that gene in the sample.

The process of creating microarrays involves attaching strands of DNA that compliments the DNA of the genes to be studied, to spots on a grid known as an array [14]. The sample is then hybridized onto the grid, allowing genes in the sample to attach to the complementary spots on the grid. A number of methods can be used to measure the level of genes attached to each spot on the grid, including radioactively or fluorescently. The abundance of genes at each spot are quantified using a laser scanner to determine the intensity.

Due to many hard to control factors involved with creating microarrays, such as chemical reactions in the sample preparation, the measured abundances of genes are not on an absolute scale, making analysis more challenging.

Using microarrays biostatisticians are able study the effect of treatments and diseases as they monitor gene expression levels, over a period of time. They can also study samples taken from a group of patients to isolate genes related to specific conditions, by comparing known healthy and diseased samples.

### 2.3.2 Statistical gene analysis

The measurements in a microarray are subjected to many different variations, due to the method of creation. These variations and the fact that microarrays can be very large

and contain complex data make analysis complicated. For example, to study the genes related to a particular medical condition, a microarray containing a sample of patients with the condition would be compared to a microarray of patients without the condition. Using statistical functions such as correlation, specific genes can be identified, relating to that particular condition.

Due to the method of creating microarrays, the analysis workflow includes a preprocessing step, before the true analysis can begin. The preprocessing includes steps to filter invalid values, quality control and normalisation of the data. These steps are very intensive and take a large amount of time to apply.

Bootstrapping techniques are used in gene analysis to asses the reliability of results. For example to confirm the results of a cluster analysis [15].

## 2.4 The System (Ness)

The HPC service provided by EPCC (Edinburgh Parallel Computing Centre) for staff and postgraduate students is called Ness [16]. EPCC follow a naming convention using the names of Scottish lochs for HPC systems they manage, Ness being named after Loch Ness.

Ness was the development platform for this dissertation with all testing and benchmarks undertaken on its backend.

Ness is a cluster of 3 physical boxes, a front end with 2 processors and 2 backend boxes with 16 processors each. All processors are 2.6 GHz AMD Opteron (AMD64e) with 2 GB of memory available.

The system runs Scientific Linux release 5, which is based on Red Hat Enterprise Linux source code, recompiled with extra features used by labs and universities.

Access to the backend is managed via Sun Grid Engine, using different queues to manage all jobs submitted to the backend.

## 2.5 The Dataset

Datasets used in gene studies contain the genes of any number of patients. The datasets and workflows that have been provided to the SPRINT teams, are regarded as of typical size for gene analysis and contain approximately 22000 genes from hundreds of different patients.

It was decided to use a random dataset in this dissertation rather than a true gene dataset. This was decided because preprocessing and applying a statistical function to a true gene dataset would have slowed development time.

For development, testing and benchmarking a random dataset was therefore created using the *sample* function of R. The dataset was created as a list of integers.

During development results could easily be verified with this integer dataset, something that would be impossible with a gene dataset.

# Chapter 3

# Parallel bootstrapping with R and parallel R packages

Before attempting to create a parallel bootstrapping implementation the serial version was analysed. With the results of the analysis described in the section below. Following that, implementations were created using two different packages available to parallelise problems in R, RMPI and SNOW. These are discussed in the following sections of this chapter.

## 3.1 Analysis of R boot function

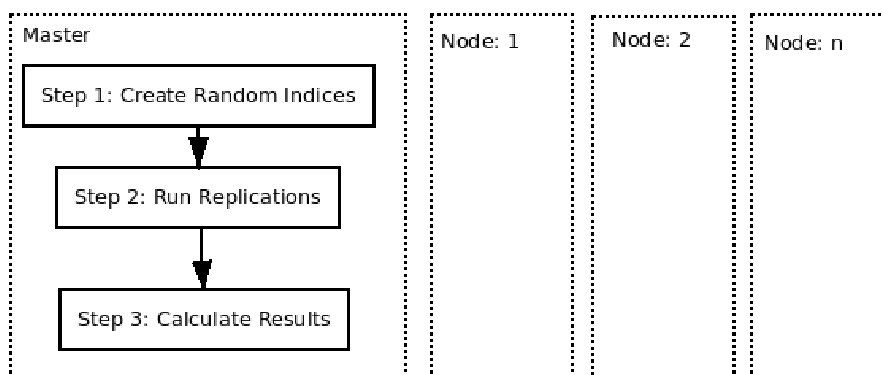


Figure 3.1: Analysis of boot() function from R boot library

Bootstrapping in R is provided by the *boot* function, which is part of the the boot library [17]. It includes support for numerous bootstrapping techniques, which are beyond the scope of this dissertation. In this project only a basic bootstrapping example with default options was considered.

The *boot* function requires three arguments for default bootstrapping. These are dataset, statistical function and number of replications. Dataset can be a vector or matrix con-

taining the data that will then be resampled a number of times, specified by the number of replications argument, to generate a set of new samples.

The statistical function is the name of a function to be called on the newly generated samples.

The statistical function passed to the *boot* function can be declared by the user and it needs to accept 2 arguments dataset and indices. The dataset argument will be the same dataset that is passed to the boot function. The indices argument will be a vector of integers the same size as the dataset. Indices are then used by the statistical function to create the resample dataset.

Execution of the *boot* function can be broken down into three steps, creating random indices, executing resamples and calculating results. This is shown graphically in figure 3.1.

In the first step, create random indices. A vector of random numbers is created using the *sample* function. The size of the indices is the number of replications that will be performed multiplied by the length of the dataset. Dimensions are then added to this vector so that it is available in a matrix format. Each row being the length of the dataset and the number of columns the amount of replications to be performed. The matrix is created in this way so each replication can easily be allocated a row, which it will then use as an indices.

The second step involves executing the resamples. A "for" loop iterates over the number of resamples that need to be performed. For each iteration the statistical function given in the arguments is called. The statistical function is called with 2 arguments, the first argument being the dataset and the second a vector called *rind* on which the resamples dataset will be based. The dataset is the complete dataset, *rind* being one row from the indices created in step one, with the row number matching the iteration number. Results of the iteration are stored in a vector called *rstat*.

The third and final step calculates and displays the results. The statistical function is performed on the original dataset to get the original result which is displayed. Using the original result and the results from all resamples stored in the *rstat* vector the bias and standard error are displayed, with a boot object being returned.

This boot object can then be used to graph the results and reused in confidence intervals calculations, these are beyond the scope of this project so ignored.

## 3.2 Random Number Generators

Bootstrapping uses random numbers to generate indices that are used to sample the original dataset, creating a new dataset. The statistical function supplied is then applied to this new dataset.

In developing versions of the bootstrapping algorithm it was important to be able to

seed the pseudorandom number generator (PRNG), so that results could be replicated and compared with other bootstrapping implementations.

Seeding the PRNG produced a number of challenges when using RMPI and SNOW, in both of these packages when a new node is spawned it inherits the random seed from the parent. Thus each node would generate the exact same sequence of numbers for resampled indices and ultimately the same results.

The PRNG can be reseeded on each node to over come this problem. However when doing this great care must be taken that the seeds are generated in such a way that there is no chance of a correlate between the random streams on different nodes.

Two methods where investigated to over come this, firstly creating all the random indices on the master then sending them to the nodes and secondly using a specialised package to deal with PRNG on threads called rlecuyer [18].

The first approach of creating all the random indices on the master then sending them to the nodes, allows the indices to be created in the exact same way as the existing boot function. This is a great advantage as the output of the parallel version is exactly the same as the serial version. A disadvantage of this approach however is the increased communication overhead of sending the indices to each node. Based on this a less elegant workaround was devised, seeding each thread with the same seed. Then on each node creating the complete indices but only using the appropriate rows from the indices based on the replications assigned to it. This approach has increased calculations on each node, but requires less communication.

Using rlecuyer to produce a stream of random numbers for each thread is a method suggest by my dissertation supervisor Michal Piotrowski, that can be used for RMPI and SNOW, however the streams it produces do not correlate with the serial streams. This means that there is no way to generate the exact results produced with the serial version. Results also vary depending on the number of streams generated, making the results harder to verify.

## 3.3 RMPI

One of the most commonly used packages to parallelise R scripts is RMPI. In this section we investigate using it to parallelise a bootstrapping problem.

RMPI [19] is a R package that provides a wrapper to MPI functions for message passing. It is not just an interface to the standard MPI functions but also provides additional functionality suited to programs written in R, including methods to create nodes and send R objects to nodes.

RMPI version 0.5-8 is documented to support MPICH [20], MPICH2[21], LAM-MPI[22] and OpenMPI[23]. However for reasons not found it would not work correctly using the default installed version 1.0.5 of MPICH2 on the ness system. No error message

was produced, the scripts would appear to spawn the required nodes successfully and then timeout without receiving any communication. OpenMPI version 1.4.2 was instead installed in a home folder on ness.epcc.ed.ac.uk with R and RMPI recompiled to use it, this did not produce the same error and worked correctly.

Due to the extra functionality RMPI provides over the standard MPI functions, R scripts using RMPI do not need to follow the traditional MPI format of having statements to perform different actions based on rank. Scripts can instead use the paradigm of master and slave. The master node spawning slaves nodes then performing actions on slave nodes and communicating via R objects.

### 3.3.1 Implementation boot-RPMI1

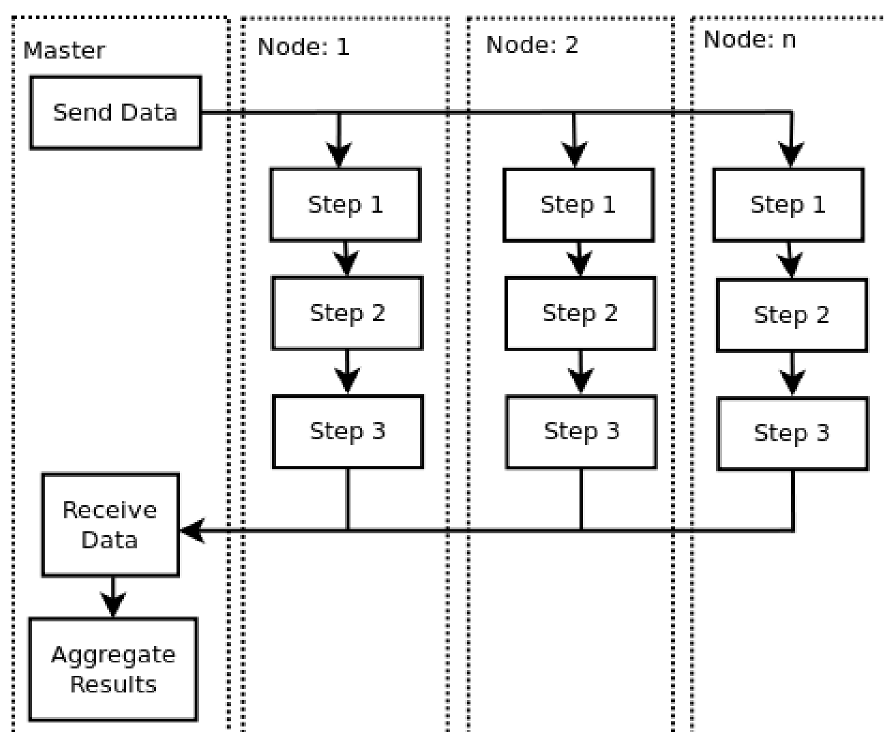


Figure 3.2: High level parallelisation of boot

The first investigation into using RMPI to parallelise bootstrapping used a high level approach, sending the function, dataset and relevant objects to each node and then executing the *boot* function remotely on each node. A graphic interpretation of this approach is given in figure 3.2, with the 3 steps as described in section 3.1. The number of replications was divided among the slaves, with the original *boot* function being called on each slave with the appropriate number of replications.

This approach was relatively simple to implement into a version *boot-RPMI1*, not requiring any specialised MPI knowledge. The speedup it provided was limited by the large size communications required to send the resulting complete boot object from

each node to the master. The output results in this investigation only required a few elements in the boot object, so sending the complete object was unnecessary. An optimised version, that did not send the complete boot object from each slave node was next looked at.

### 3.3.2 Implementation boot-RPMI2

The next version to be implemented was *boot-RMPI2*, instead of calling *boot* directly on each slave, a wrapper was executed. The wrapper function executing the *boot* function storing the results in an object. Portions of the object that are required to perform the calculation of the standard deviation are then sent to the master. Sending just part of the object back to the master instead of the complete object, significantly reduced the size of the messages sent to the master. Speedup provided by this solution was the best of all the methods investigated using SNOW and RMPI.

The *boot-RMPI2* implementation of the script is more complex, with the developer needing to handle the communication between the master and worker processes, hence a better understanding of MPI is required. The SPRINT user survey identified the majority of users as beginners or having no experience with parallel programming. For users with limited HPC and parallel programming experience, implementation of this level of script would be more challenging.

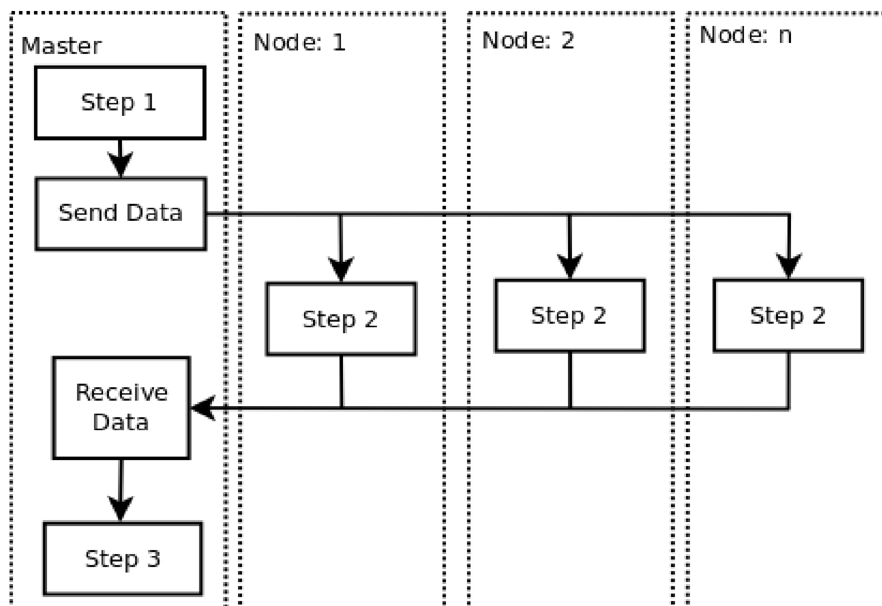


Figure 3.3: Low level parallelisation of boot

### 3.3.3 Implementation boot-RPMI3

Further investigation involved a parallel implementation of a low level version of the bootstrapping function. The motivation to investigate a low level implementation was that it allowed the random indices to be created on the master in the same way it was done in the standard serial version, thus being able to get the exact same results from the parallel version and the serial version.

Figure 3.3 depicts this low level approach, with the 3 steps described in section 3.1. The master performs step one, creating the random indices in exactly the same way as the serial version. The indices are then sent to each slave node. These then performs step two, processing their replications and sending the results back to the master. The master then performs step three, calculating the results.

Initially the implementation *boot-RMPI3* based on this method proved ineffective as the time taken performing communications to send the random indices to each node negated any speedup. Once this bottleneck was identified an optimised version was designed that instead created the exact same complete indices on each node. This meant duplicating the same calculates on each node to create the exact same indices but reduced communication drastically as the indices did not need to be sent to each node. This resulted in much better performance and speedup.

This design is depicted in figure 3.4, with the creation of complete indices being performed in step 1 on each node. Then step 2 with each node performing the appropriate replications. With the results of the replications being sent back to the master which performs the final step 3.

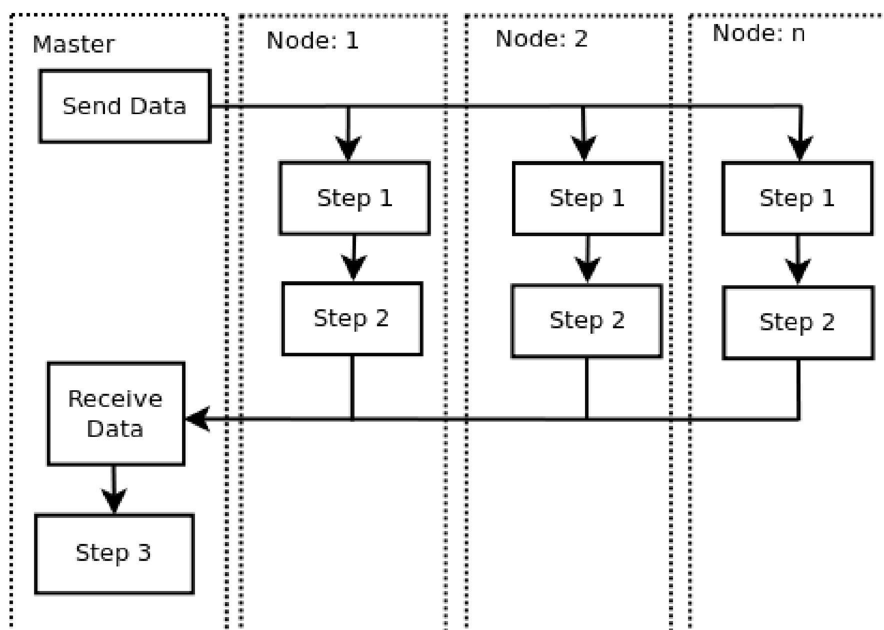


Figure 3.4: Optimised parallelisation of boot

## 3.4 SNOW

The SNOW package is very similar to RMPI, providing similar methods to parallelise R scripts and using the same principles. In this section SNOW is used to implement bootstrapping using the design pattern as used for RMPI in the section above.

SNOW [24] is a package that wraps a number of communication protocols and provides simple functions to make use of these inside R. Using an abstract form of master and slave nodes allow R scripts to be developed independently of the underlying communication layer. Also allows the communication layer to be replaced when needed, thus making the scripts more portable.

The library also provides functions that make implementation easier than using the native calls of the underlying communication library.

The SNOW interface provides access to the communication libraries MPI, PVM and Sockets. The interface offers a limited amount of functions that allow simple parallelisation of problems. Complex communication cannot be achieved, as decomposition can only be done using the functions that are available.

It does not provide the functionality for nodes to send messages to other nodes or for calculations to be performed based on a nodes unique rank. The provided functions focus on performing operations on a matrix in parallel or parallelisation of loop structures.

In the bootstrapping implementation an attempt was made to parallelise at a low level, dividing step 2 among the nodes. It was implemented using the *clusterApplyLB* function of SNOW to divide the replications between nodes. The *clusterApplyLB* function spreads the work load among the slave nodes in a balanced way.

This method proved slow, providing no speedup. This was due to the large number of communications that needed to be sent. For each replication a number of small messages needed to be sent, this made the communication cost too large to provide any speedup.

The next implementation *boot-SNOW* investigated parallelising bootstrapping at a higher level, calling the *boot* function on each node and receiving the results back on the master. The master then aggregates the result objects so that the output was in the same format as the serial version.

The speedup achieved using this approach was not impressive, due to the large amount of time required to send the results back from each node. The results are stored in a boot object, the size of which increases depending on the size of the dataset and replications. The information required for the output does not require all the elements of the boot objects, so the sending of the complete object creates unnecessarily large communications.

It would be possible using SNOW to write a more effective method to parallelise bootstrapping, most likely writing a wrapper function that simply returned aspects of the

boot object that are required for the output results. However doing this does not make any use of the features provided by SNOW and could be more simply implemented using RMPI.

## 3.5 Speedup of RMPI and SNOW implementations

The performance of RMPI and SNOW implementations of *boot* are illustrated in figure 3.5. To created these graphs, a number of benchmarks were run for each of the implementations on 2,4,8 and 16 processors, with dataset of 3000 integers and 30000 replications. The speedup of each implementation calculated against the execution time of the serial version of *boot*.

Speedup is the standard method of describing the increase of performance a parallel version of a program obtains over its serial version. It is calculated by dividing the execution time of the parallel version on a particular number of processors by the execution time of the serial version.

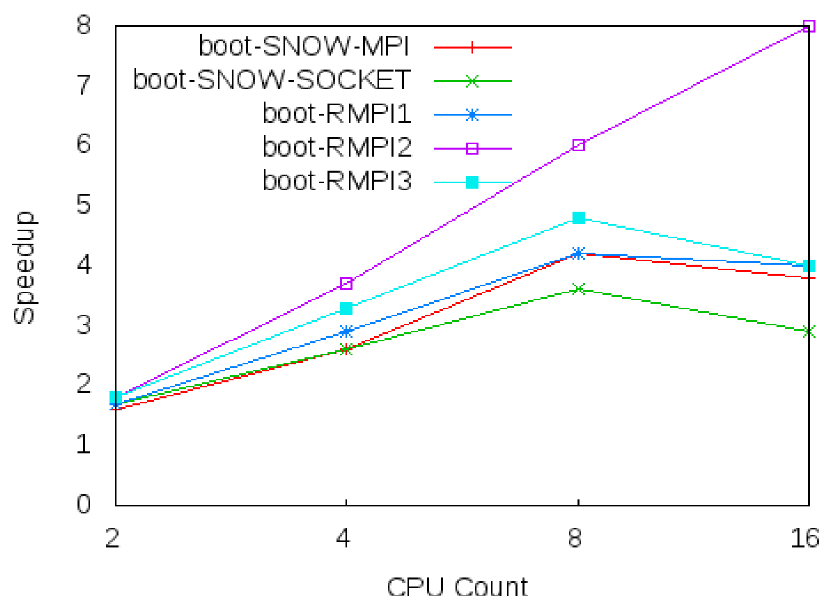


Figure 3.5: Speedup of R parallel methods with boot

To compare the performance of MPI and Sockets the *boot-SNOW* implementation was run with both underlying communication layers. Up to 4 processors the results where very similar but at 8 processors MPI provided more speedup and continued until 16 processors. MPI is a specialised library for message passing with optimisations to allow message passing on shared memory systems such as ness. Thus it would be expected that it out performed the socket based method.

The implementation *boot-RMPI2* offers the most speedup on each number of processors and the speedup appears to continue. This is due to the optimizations made to it

reducing the size of the result communications sent back to the master from the slave nodes.

All other implementations show peak speedup at 8 processors, after this point the overhead of the communications negates any improvement in speed that is made by calculating replications on more nodes.

# Chapter 4

# Writing extensions to R in C

The R scripting language and packages are very extensible, however on occasion a problem will need to be implemented in another language and interfaced with R. The SPRINT framework is an example of this, it needs to make use of the MPI library for message passing. MPI is available as a library for a number of programming languages including C, Fortran and C++.

SPRINT is written in C and thus uses MPI libraries and the R API for C.

Before functions can be developed using SPRINT, the R API needs to be understood. The sections below discuss the R API and how it can be used to evaluate R expressions from C.

## 4.1 The R API

As R is written in C, all the functions that are used internally in R are available along with methods to call C functions from inside R and the ability to call R functions from C.

To make use of these R functions in C, the header files R.h and Rinternals.h need to be included in the C source code.

The R API has an online manual "Writing R Extensions" [25] with a few limited examples. This manual did not go into enough detail for the functions required in this project. So the Rinternals.h header file was used as the resource for the API, with usage of functions determined based on name, arguments and return values.

All variables in R are stored as objects called SEXP. These objects are identified by pointers and need to be accessed via functions and macros provided in the R libraries. This caused some challenges in development, as functions listed in Rinternals.h have return values or arguments of type SEXP. Internally these require a specific type of

SEXP object and give errors when called with the wrong type of object. These errors only occur at runtime and not during compilation making development more complex.

## 4.2 R objects

R objects in C are of the type SEXP [26]. They are a pointer to memory which contains a structure managed by R. Management of memory associated with R objects is done by the garbage collection in R, as described in the following section.

Objects in R can be of many different types, the most up to date reference being the Rinternals.h header file. Each object type is identified by an integer and can be found using the *typeof* function from the R API.

Any variable in R is stored as an object and can be passed from R to a C function by calling the R function *.Call*. The *.Call* function takes one or more arguments, the first the name of the C function to call and all proceeding arguments SEXP objects that the C function will be called with.

The most common types of objects used in this project are REALSXP, INTSXP and LANGSXP. REALSXP and INTSXP are vector objects containing pointers to integer or double values. LANGSXP objects contain a language which can be evaluated inside R, the first element a reference to a function and the remaining elements the values to be called with that function as arguments. The evaluation of LANGSXP objects is further discussed in section 4.4.

To get the address of a object that has not been passed to the C function and is available in the R parent process. The *install* function needs to be used, called with a string identified the object name it returns a pointer to the R object. Allowing R objects that have been created in R to be used in C without having to pass the object over the interface. This is very effective in that interface does not need to transfer a large number of objects and that objects do not necessarily have to be converted when passed between R and C.

## 4.3 Garbage collector

Memory management in R is done automatically, functions that create objects and initialise vectors trigger allocation of memory. Memory is freed from unused objects by the garbage collector.

The garbage collector runs occasionally and frees memory allocated to objects that are not linked to variables stored on stack anymore. Two macros *PROTECT* and *UNPROTECT* are available to signal to the garbage collector if a object is in use or if it can be freed. The *PROTECT* macro protects an object so it will not be freed by the garbage

collector. An object needs to be protected before any other operation occurs that may allocate memory in R, or else the object maybe overwritten in memory.

R objects that are passed from R to C are already linked to variables stored on the stack in R and so do not need to be protected. Variables created in C are not referenced at R level so need to be protected.

The *UNPROTECT* macro takes an integer as argument, which signals the garbage collector to free that number of objects that are no longer used.

## 4.4 Calling the R evaluator from C

Being able to execute a function defined in R from C is critical to implementing bootstrapping in SPRINT. All SPRINT functions that have been implemented up until now rely on performing all necessary calculations in C and then returning the results back to R. The boot function however allows a user to specify a statistical function to use on resamples, this function is defined by the user in R and thus cannot be hard coded in C. To implement a bootstrapping solution an expression would need to be evaluated in R from C.

The R API provides a function *eval* which takes an expression, evaluates it inside R and then returns the results as a SEXP object. The expression passed to *eval* needs to be in SEXP object of the type LANGSXP. The LANGSXP object is a list structure of a similar style to LISP lists. The basic structure is that the first element of the list is returned from the *CAR* macro, with the next element accessible by calling the *CDR* macro. Using these two macros the list can be navigated and modified as needed.

A simpler method to create the expression is to use the *lang* functions, which return LANGSXP objects. The *lang*, *lang2*, *lang3* and *lang4* functions take the number of arguments indicated by the number at the end of the functions name. With the first argument being the function and all following being arguments to that function. Every argument to the lang functions needs to be of the type SEXP.

# Chapter 5

# The parallel pboot( ) implementation in SPRINT

Based on the knowledge gained from this projects implementations in RMPI and SNOW a parallel version boot was developed using the SPRINT framework. In this chapter the implementation of this pboot function is looked at.

The first section explains how the SPRINT framework transitions from an R script to the C functions that contain the implementation.

## 5.1 SPRINT interface to pboot

A *ptest* function is provided with SPRINT, used for testing to ensure SPRINT is working correctly and as a template for other algorithms to be implemented. All implementations of *pboot* where based on *ptest* using the same style of variables, function names and layout of files. So that this code would be consistent with the overall SPRINT project.

The pathway from the calling R script to the implementation is illustrated in figure 5.1, showing how the relevant files are interpreted and called.

The SPRINT framework is called from inside an R script by loading the sprint library. All function calls listed before the position where the library is loaded, are executed on all the nodes created by the mpiexec command. Directly after the SPRINT library is loaded the master carries on execution with the slaves in an infinite loop waiting for a command code to be broadcast from the master.

The fact that all nodes run the same R script until the SPRINT library has loaded, means that all objects defined before the loading of the SPRINT library are available on all nodes. This is a notable difference from the way RMPI and SNOW work and reduces the required communications as objects can be created on all nodes before calling the SPRINT library. It proved useful in the bootstrapping implementations as the dataset

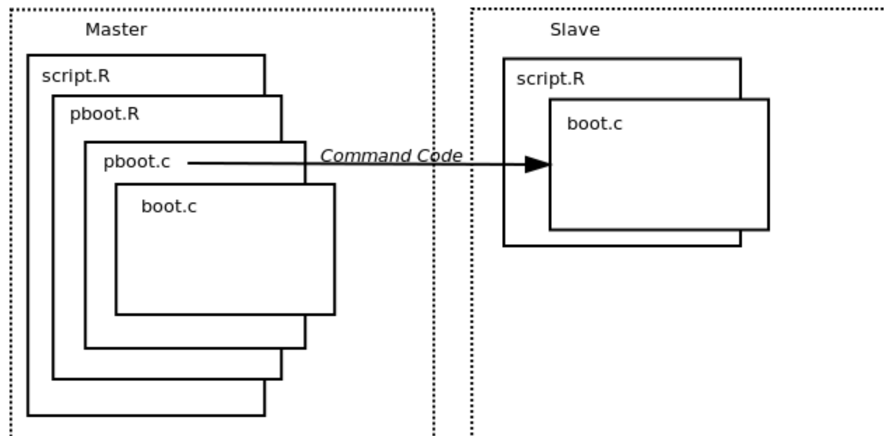


Figure 5.1: SPRINT framework structure

was loaded and the statistical function defined before the SPRINT library was loaded. This meant that in the SPRINT implementation the dataset and statistical function did not need to be sent to all nodes, only the names identifying them. Reducing the size of communications performed in the SPRINT implementation.

A *pboot* function is defined in an R script pboot.R, this function acts as the interface to the C functions. This function is only executed on the master after the SPRINT library is called. Inside the *pboot* function a number of operations are performed before the C interface function. Most importantly the original object name of the passed variables are determined.

When the *pboot* function is called from the R script the arguments it is passed are renamed inside the function according to the definition of the *pboot* function. Due to this the original name of the object, and how it can be accessed on slaves, is lost. To overcome this the R function *substitute* was used, it returns the original name associated with the object before it was passed to the function. Having this information allows these objects to be used on slave nodes by just sending the objects name, rather than sending a complete object which would be larger. As each node has ran the same script up until the point when the SPRINT library was loaded they each have the same objects with the same names available, allowing this scheme to work.

The *pboot* R function now calls the C interface function, defined in the pboot.c file, and passes its arguments as SEXP objects. When the C interface function executes on the master it sends a broadcasts to all nodes. This broadcast sends the command code that the slave nodes are waiting for. The command code is associated with an implementation function that is called by the slaves. In this example the implementation function is located in the boot.c file.

The implementation function is defined as having a variable number of arguments. This allows it to be called on the master with all the arguments passed from R, but on slaves where there is no access to the arguments, it is called with no real arguments. One of the arguments that the master receives is a pointer to an array where the results are to

be put. In this way results are passed from the implementation back to the interface via the array listed in the arguments.

Once the implementation function has been completed on the slave nodes, they return an integer status code without processing any more of the calling R script. On the master the implementation function returns back to the interface function.

The interface converts the results stored in an array into a SEXP object which is then returned back to the R function on the master. The R function then processes the returned SEXP object and displays the results.

## 5.2 Parallel strategy

The serial *boot* function has one "for" loop, described as step two in the section 3.1. This is where the majority of time is spent during execution and the obvious operation to parallelise.

### 5.2.1 Implementation pboot1

The first strategy investigated in implementation *pboot1* involved the master process calculating the indices then sending each slave the segment of the indices it was assigned to process.

Each slave would wait to receives its own segment of the indices, once received the slave runs its replications based on this indices, then returns an array to the master. The master would receive all the result arrays from the slaves and place the results in an SEXP object which is returned back to R. This strategy is the same as depicted in figure 3.3. The pseudo code for this implementations is in listing 5.1.

```
1   if (rank == master) {
2     create_indices()
3     decomposition()
4     send_indices()
5   } else {
6     receive_indices()
7   }
8
9   for replications in decomposition {
10   statistical_function()
11  }
12
13  if (rank == master) {
14   receive_results()
15  } else {
16   send_results()
17  }
```

Listing 5.1: Pseudo Code of pboot1

### 5.2.2 Implementation pboot2

The second approach called *pboot2*, aimed to illustrate the most effective implementation, but with a caution of its statistical accuracy due to the possibility that the PRNG streams may not be independent.

This implementation would not give the exact same results as the serial versions, as the PRNG could not be seeded in exactly the same way. The results also varied according to the number of processors the decomposition was based on.

The strategy was for each node to be assigned its own number of replications to perform, then as it performed each replication to generate the random numbers required to create an indices. After each node completing its replications the results are sent back to the master in the same way as in the first strategy. The pseudo code for this implementations is in listing 5.2.

```
1   decomposition()
2   for replications in decomposition {
3     single_indices()
4     statistical_function()
5   }
6   if (rank == master) {
7    receive_results()
8   } else {
9    send_results()
10  }
```

Listing 5.2: Pseudo Code of pboot2

### 5.2.3 Implementation pboot3

The third strategy *pboot3*, was an attempt to optimise the performance of *pboot1*. The performance in *pboot1* was limited by the speed at which the indices could be sent to each node. It was surmised that the time taken to communicate the indices to each node from the master was more than it would take to calculate the complete indices on each node.

So the strategy in *pboot3* involved each node creating an identical complete indices. Then processing just the replications from the indices that the decomposition had assigned to it. Before sending the results back to the master. This strategy is the same as depicted in figure 3.4 and used with *boot-RMPI3*. The pseudo code for this implementations is in listing 5.3.

```
1   create_indice()
2   decomposition()
3   for replications in decomposition {
4    statistical_function()
5   }
6   if (rank == master) {
7    receive_results()
8   } else {
9    send_results()
10  }
```

Listing 5.3: Pseudo Code of pboot3

## 5.3 Work distribution

Due to the fact that bootstrapping can be called with any statistical function and the way that sample datasets are created using random numbers, there is no way to guarantee an even distribution of work load between nodes. The execution time of each replication will vary based on the contents of the dataset that the statistical function is called against. For example if the original dataset was a particularly sparse vector, random generation of datasets for different replications would result in some replications having very sparse datasets which some statistical functions will be able to calculate quickly. Other randomly generated datasets would have more values and thus take longer to process by the same statistical function.

In all the implementations of this project the number of replications are divided equally among the nodes. Thus each node will call the same statistic the same amount of times. Each replication on each node will however have different datasets to process and this could lead to an uneven workload.

The datasets on each node are based on random numbers and the characteristics of the statistical function cannot be known so, ensuring an even work distribution using static decomposition is impossible.

Using a form of dynamic decomposition would be possible, with nodes being able to assist other nodes if they have already completed their assigned replications. This would however only be useful if boot was called with a dataset that was very sparse or particularly unbalanced with some replications taking significantly more time than others. Investigation into this was beyond the scope of this project and it was assumed that every replication would take roughly the same amount of time.

## 5.4 Communication

Communication in SPRINT is done using MPI. Once the *pboot* function is called the communication is initiated by the master broadcasting a command code to all nodes. This command code is associated with an implementation function which is then called on the slave.

After this communication, all the information needed by the slaves to perform the replications is sent via broadcast, including a string with the statistical function's name and the string with the name of the dataset. These communications will always be small as they are only strings containing the names of objects not the full object.

The names of the function and dataset is sent to slave nodes rather than complete R objects, to reduce the size of communications. As each node has processed the R script up until the SPRINT library was loaded, they will already have the R objects in memory.

In the *pboot1* implementation, each node are sent a segment of the indices that it has been assigned during decomposition. This can be a large communication as the size of the indices to be sent is $\frac{\textit{number of replications} \times \textit{length of dataset}}{\textit{number of processors}}$. As such an increase in either replications or dataset size will increase this communication and an increase in number of CPUs will decrease the size.

The implementations *pboot2* and *pboot3* avoid sending the indices to other nodes and so save significantly in communication.

The results are sent back to the master in an array of doubles. Each nodes array of results is $\frac{\textit{number of replications}}{\textit{number of cpus}}$. Thus scales with number of replications and CPUs.

## 5.5 Testing

The requirements we set for a parallel version of boot was that it was stable and that it produces accurate, reproducible results. The results could be compared to the original serial version of boot that is standard in R, to ensure the integrity of the results.

The various aspects of testing are discussed further in the below subsections.

### 5.5.1 Component testing

During development different segments of code were tested to ensure they worked as expected. Generally these tests would be performed with a smaller dataset and the results compared to the same segment of the serial version of the code.

The particular sections that were tested in this way were replication decomposition, indices creation, indices distribution and replication results.

These component tests were done informally using print statements to display C variables and the *PrintValue* function for R objects. These tests were removed after the component was verified to work correctly.

This method of testing was particularly useful when getting the parallel version to produce the same indices as the serial version, by displaying the matrix allowing them to be compared. Likewise the array holding the results of the replications was also displayed to compare with the same array in the serial version to ensure the parallel versions replications had the same results and in the same order.

### 5.5.2 System testing

As implementations *boot-RMPI3*, *pboot1* and *pboot3* created the random indices in the same way as the serial version of boot, by seeding the PRNG it was possible to get the exact same results as the standard version of boot. This made testing of these implementations straight forward, with the results they produced checked against the serial version to confirm the results.

Other implementations used different methods to create the random indices including a library to produce streams of random numbers. These implementations where a challenge to test as the results differed from the serial version of boot, the results from implementations using streams of random numbers also varied depending how many streams where used. As each node uses a different stream this meant that results produces when run on different numbers of processors also varied.

To try and better understand the problem a graph was plotted, that is represented in in figure 5.2. The standard error result from three different implementations using different methods to created indices are compared. All of these executions where run on 8 processors, with a problem size of 3000 integers.

The three implementations used illustrate the different methods used to create indices in this dissertation. Results from the serial boot version where identical to versions *boot-RMPI3*, *pboot1* and *pboot3*. Implementation *pboot3* used a the standard PRNG in R and naively seeded each node based on rank, which is warned may cause inaccurate results due to random number streams not being fully independent. The remaining RMPI and SNOW implementations used the rlecuyer library, that produces independent streams of random numbers.

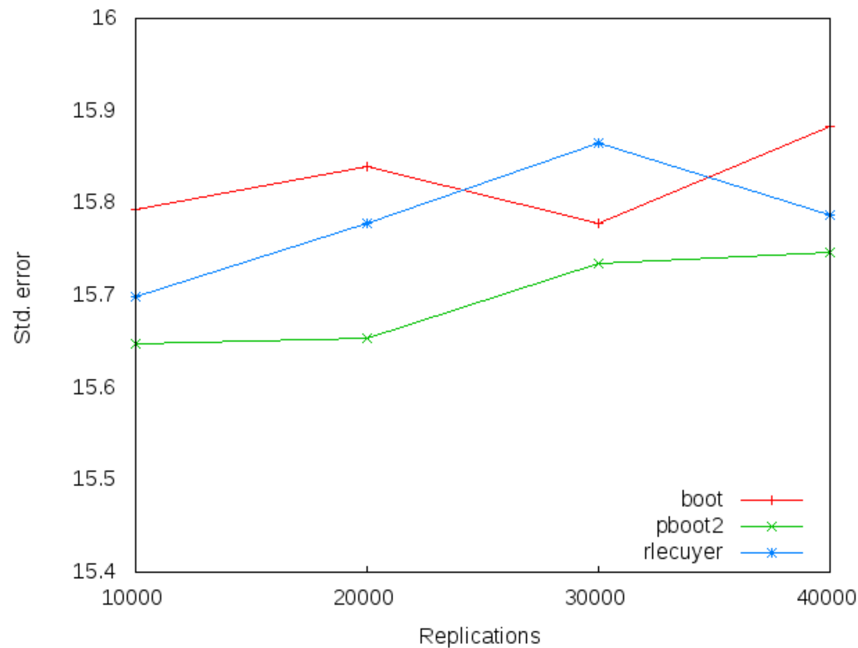


Figure 5.2: Standard error with various implementations

As bootstrapping results are estimates and based on random numbers it is not possible to define a range in which the results need to be within to be deemed correct. From the results it can be seen that all implementations produce very similar results that vary in a similar range. Indicating that the results are valid.

### 5.5.3 Stability testing

During development and benchmarks all version where run multiple times. Due to the PRNG being seeded the results from each run where identical. At no stage did the software crashed or gave errors. These indicate the software is stable, with this particular dataset and statistical function.

Before being released as a stable implementation further tests should be undertaken to ensure that it remains stable with more complex datasets and statistical functions.

# Chapter 6

# Results and performance analysis

To measure the performance of the various implementations of parallel bootstrapping function in SPRINT a number of benchmarks were undertaken. These benchmarks were run on the backend of the ness system. The results discussed and represented in the below sections of this chapter.

The four factors that effect the execution time of the bootstrapping algorithms are dataset size, number of replications, complexity of the statistical function and number of processors. Each of these factors will be discussed and compared for the three implementations of *pboot* in sections below.

## 6.1 Speedup of pboot() implementations

The results in figure 6.1 are calculated against the execution time of the standard *boot* function, a dataset of 3000 integers over 30000 replications.

All versions show very good speedup, particularly on 2 and 4 processors, where the speedup is nearly linear with the number of CPUs. This indicates that all versions would be ideally suited for SMP machines which commonly have dual or quad core processors. On 8 and 16 processors *pboot2* and *pboot3* show the best speedup but are no longer able to offer linear speedup with the increase of CPU.

Implementation *pboot2* provides the biggest speedup over 8 and 16 processors, this is due to the reduced communications in the design. Unlike *pboot1*, *pboot2* does not create the random matrix on the master and then send decompositions of this matrix to slaves. Instead *pboot2* creates the indices on each node as they are needed for each replication. Hence has less communication and faster execution.

The speedup results for *pboot3* are the most useful as this implementation, like *pboot1*, produces the exact same results as the serial version of boot. So this version can be put forward as beta software to be used for bootstrapping.

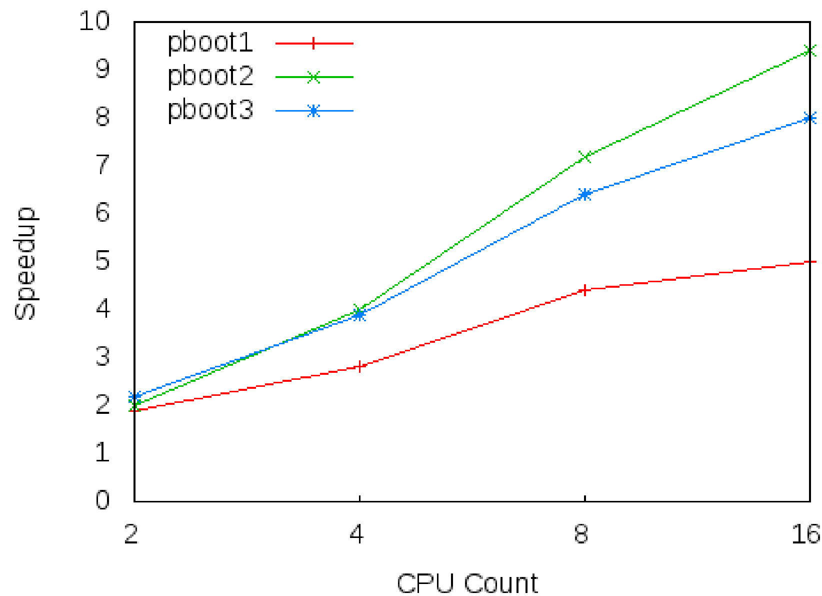


Figure 6.1: Speedup of pboot

## 6.2 Scaling of pboot() implementations with different problem sizes

In figure 6.2 the effect of the size of dataset on the execution time of the implementations are compared to execution time of the serial version. All these benchmarks where performed on 16 processors with 30000 replications.

The speedup of *pboot1* remains steady with the increase of datasize, this is due to the larger communication this implementation is required to send with a increased size of dataset. This larger communication takes longer and so the speedup remains constant.

In implementation *pboot2* and *pboot3* size of the communications does not change with an increase size of dataset. So the communications will remain the same size and take the same amount of time. This allows them to offer a greater speedup when compared to the serial version.

The *pboot2* implementation consistently offers the best speedup in these benchmarks. This is due to only creating the indices it requires for the replication it is doing. While version *pboot3* , each node creates a full indices but only uses the rows applicable to the replications it undertakes.

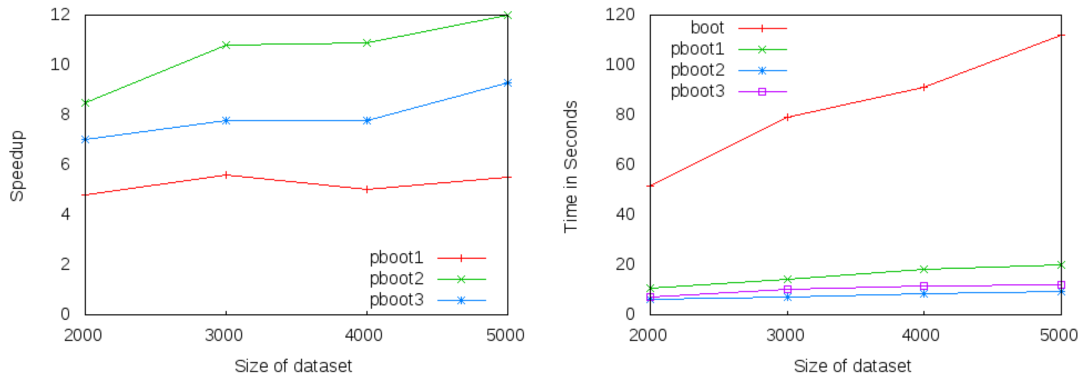


Figure 6.2: Scaling of pboot based on problem size

## 6.3 Scaling of pboot( ) implementation with different number of replications

Figure 6.3 shows the effect of number of replications on the three *pboot* implementations compared to the serial version of boot. All these benchmarks where performed using 16 processors with a dataset size of 3000.

As in the above section the speedup provided by *pboot1* remains stable, this is once again due to the increase in replications effecting the size of the indices that needs to be sent to each of the slave nodes. The increase time spent performing communications stopping any extra speedup being achieved.

Both *pboot2* and *pboot1* improve speedup until 8 processors after which the speedup remained roughly consistent till 16 processors. With the increase of replications, the amount of work each node needs to perform increases as does the size of the result arrays that are sent back to the the master. The increase in the size of the results array that is sent back to the master would result in slower communications, limiting any improvement in speedup.

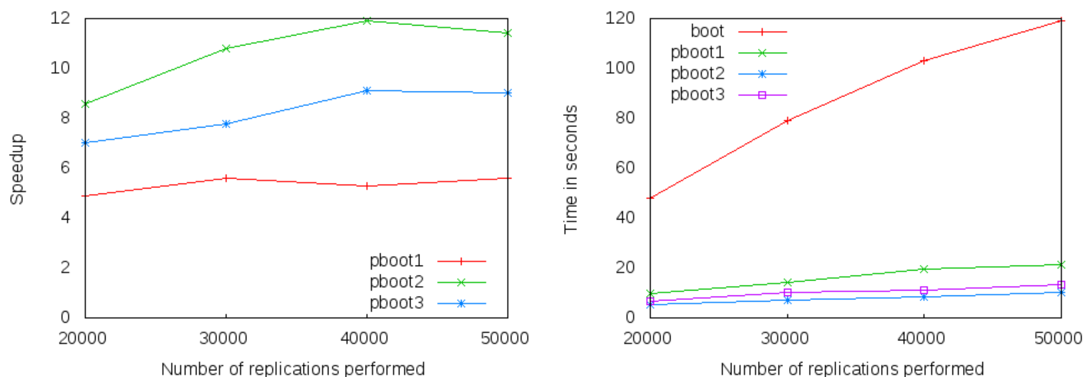


Figure 6.3: Scaling of pboot based on replication size

## 6.4 Scaling of pboot( ) implementation with complexity of statistical function

To analyse how the implementations of *pboot* would perform when called with statistical function of different complexity, benchmarks were undertaken on 16 processors, with 30000 replications and dataset of size 3000. To simulate a statistical function that was two, three and four times as complex as the original, the function was called the appropriate number of times.

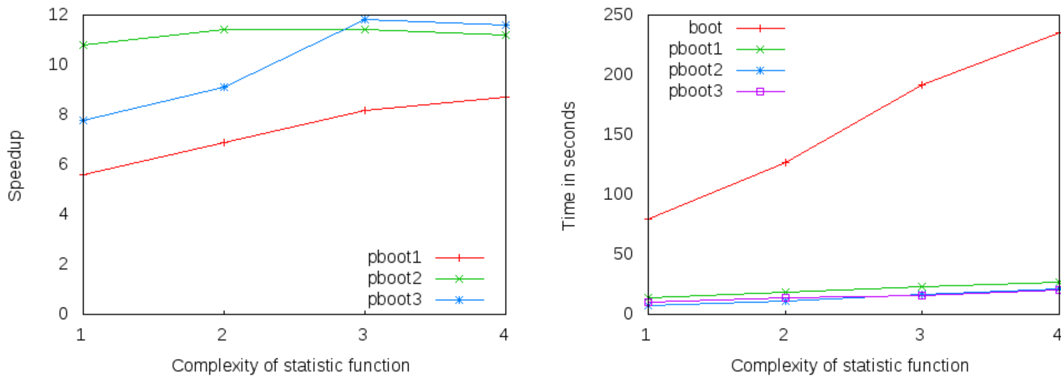


Figure 6.4: Scaling of pboot based on problem complexity

The results in fig 6.4 show that *pboot1* was able to improve speedup with a more complex statistical function. This is due to the size and number of communications staying the same in increased statistical function complexity. While the speed at which the replications are processed is increased compared to the serial version. However this implementation will never achieve the same speedup as *pboot3*, due to the fact that while the master creates the indices, the slave nodes are sitting idle and cannot continue until they have received the inices. In *pboot3*, each node creates a complete indices and does not need to wait for any communication before it starts processing its replications.

Implementation *pboot3* shows good improvement in speedup, with slightly more speedup than when run with the statistical function of three and times complexity. It is suspected that *pboot2* is able to offer better speedup in this scenario due to only calling the sample function once, while version *pboot3* needs to call the function for every replication, which maybe slightly less effective and thus take longer.

## 6.5 Comparison of speedup with RMPI and SNOW implementations

In figure 6.5 the speedup results for all the implementations developed in this project were plotted, to compare the relative performance of SPRINT, SNOW and RMPI.

Up to 8 processors all implementations show the same characteristic of improved speedup. After that point the speedup of implementations that rely on sending the indices to slave nodes, stays the same or drops slightly. This is due to the increase in number of communications that needs to be sent, which take longer to perform. Reducing the advantage of having less replications to process on each node.

The three implementations *boot-RMPI2*, *pboot3* and *pboot2* that show continued increased speedup after 8 processors all do not send the indices from the master to slaves. Avoiding this communication, results in the continued speedup. Both *boot-RMPI2* and *pboot2* avoid this by creating the row of each indices as required when performing a replication, this is the most effective method but makes it impossible to get the same results as produced in the serial version.

Implementation *pboot3* avoids the communication by creating the full indices on each node in exactly the same way as the serial version. Allowing the results to be identical to the serial version without any costly communication.

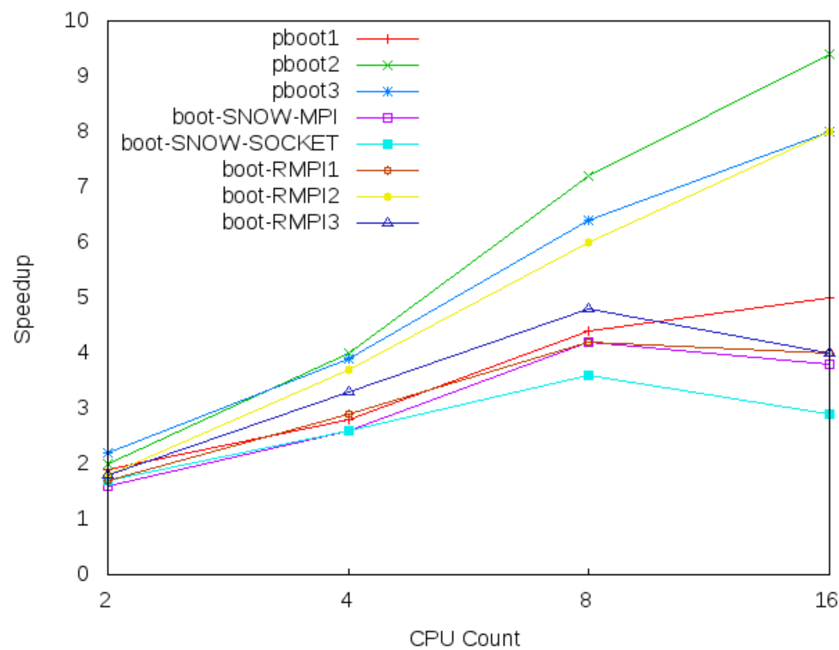


Figure 6.5: Comparison of speedup of SPRINT, RMPI and SNOW implementations

# Chapter 7

# Conclusion

Both RMPI and SNOW proved relatively simple to develop with. The fact that the functions where available inside the R scripting language meant that no compilation was necessary and debugging was easy. This made the development process quicker, as did the fact that both SNOW and RMPI allowed messages to be sent that contained R objects. This is not possible in SPRINT which requires R objects to be converted to standard C data types before they can be sent using MPI.

RMPI and SNOW were very similar in that they required you to follow a paradigm of a master sending tasks to a cluster of slaves. There were two notable differences that were identified between RMPI and SNOW during this projects developments. The first that SNOW allowed the underling message passing library to be changed very easily. This allowed scripts to be more portable as they could be run on a system with any of the supported message passing libraries and not just MPI. The performance of different message passing libraries could very easily be compared with the same script.

Secondly RMPI allowed you to communicate directly with individual nodes, rather than just sending broadcast commands. This level of access was essential for development of a version that produced the same result as the serial version of boot.

Despite the ease of development with SNOW and RMPI, a user with no knowledge of the principles of message passing would not easily be able to develop scripts similar to the examples in this project. The main challenge was getting the individual results from the slave nodes and aggregating them into a standard output that was expected by the user.

The development of pboot implementations for SPRINT was challenging and required knowledge of R, C, MPI and the SPRINT framework. Using the ptest function as a template, acted as a good base for development. The biggest challenges being the conversion of R objects due to the lack of documentation of the R API in particular the Rinternals.h header file.

For an end user to use any of the parallel boot implementations developed would require the same amount of work. Simply including the necessary package and calling the

parallelised version of boot rather than the serial version.

All versions developed scaled well and offered speedup, showing that all the methods are effective for parallelising bootstrapping problems. The best speedup was provided by SPRINT implementations, showing it to be the preferred method when the best speedup is required.

The biggest factor limiting speedup is the creation and communication of a single indices of random numbers. This is done so the same results are produces with parallel version as the serial version and because of concerns of the independence of PRNG streams. To avoid this in RMPI and SNOW versions a PRNG library was used that produced independent streams of random numbers. This is not available in SPRINT yet.

## 7.1 Further Work

The fastest implementation developed was *pboot2*, this cannot be put forward as a valid solution as the PRNG used cannot be guaranteed to produce independent streams of random numbers on each node. PRNG packages are available that can produce independent streams of random numbers on each node and were used in both the RMPI and SNOW versions. If such PRNG package could be made available in SPRINT then *pboot2* could be put forward as a valid solution.

In this project only the simplest default options of bootstrapping were looked at. The serial version in R is able to call functions with variable number of arguments and includes various options to the way in which bootstrapping is performed. None of this the versions developed for this project can do this at present.

The serial version returns a boot object along with the results, the boot object can be used to display the results in a graphic format and calculate confidence intervals. The parallel version cannot do this at present, but could be extended to include this functionality.

The functionality of SNOW allowing the underlying communication layer to be changed allowed code to be more portable. A method of abstracting communication in SPRINT, to allow changing the underlying communication layer could be investigated. Taking away the reliance of SPRINT on MPI.

The biggest challenge while developing the SPRINT implementations was converting R objects to native C types so they could be sent using MPI. In both SNOW and RMPI this was not necessary as messages could be sent containing R objects. If R provided wrappers so that R objects could be sent directly using MPI, development of new functions would be easier and less error prone.

The benchmarks presented in this project were compiled on the ness system which is a shared memory machine. It would be interesting to run the same series of benchmarks

on a cluster of machines and MPP systems. Both of these systems types would have a larger number of processors but typically slower interconnections.

# Appendix A

# Results from benchmarks

## A.1 Speedup

Results of speedup benchmarks performed on all implementations with dataset of 3000 integers and 30000 replications. The time in seconds for each execution are in table A.1, the calculated speedup results are in table A.2. The results of these benchmarks were used in figures 3.5, 6.1 and 6.5.

| CPUs | boot | pboot1 | pboot2 | pboot3 | boot-SNOW-MPI | boot-SNOW-SOCKET | boot-RMPI1 | boot-RMPI2 | boot-RMPI3 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 72 | 36.9 | 34.7 | 32.5 | 43.7 | 42 | 42.1 | 38.5 | 38.4 |
| 4 | 72 | 24.9 | 17.8 | 18.4 | 26.7 | 26.8 | 24.3 | 19 | 21.4 |
| 8 | 72 | 16.1 | 10 | 11.1 | 17.1 | 20 | 17 | 12 | 14.8 |
| 16 | 72 | 14.4 | 7.6 | 9 | 18.7 | 24.3 | 17.9 | 9 | 18 |

Table A.1: Execution time in seconds vs number of CPUs

| CPUs | pboot1 | pboot2 | pboot3 | boot-SNOW-MPI | boot-SNOW-SOCKET | boot-RMPI1 | boot-RMPI2 | boot-RMPI3 |
|---|---|---|---|---|---|---|---|---|
| 2 | 1.95 | 2.07 | 2.22 | 1.65 | 1.71 | 1.71 | 1.87 | 1.88 |
| 4 | 2.89 | 4.04 | 3.91 | 2.70 | 2.69 | 2.96 | 3.79 | 3.36 |
| 8 | 4.47 | 7.20 | 6.49 | 4.21 | 3.60 | 4.24 | 6.00 | 4.86 |
| 16 | 5.00 | 9.47 | 8.00 | 3.85 | 2.96 | 4.02 | 8.00 | 4.00 |

Table A.2: Speedup vs number of CPUs

## A.2 Testing

The figure 5.2 was created using the results of the executions listed in table A.3. All executions in this test were run on 8 CPUs with dataset of 3000 integers.

| Replications | 1000 | 2000 | 3000 | 4000 |
|---|---|---|---|---|
| Serial | 15.79294 | 15.8395 | 15.77745 | 15.88334 |
| pboot2 | 15.64717 | 15.65278 | 15.73512 | 15.74626 |
| rlecuyer | 15.69849 | 15.77737 | 15.86491 | 15.78635 |

Table A.3: Standard Error result with number of replications

## A.3 Problem Size

The figure 6.2 was created using the results of benchmarks executed on 16 CPUs with 30000 replications listed in table A.4 and A.5.

| problem size | 2000 | 3000 | 4000 | 5000 |
|---|---|---|---|---|
| boot | 51.5 | 79.2 | 91.2 | 112 |
| pboot1 | 10.6 | 14 | 18.2 | 20.15 |
| pboot2 | 6 | 7.3 | 8.3 | 9.3 |
| pboot3 | 7.3 | 10.1 | 11.6 | 12 |

Table A.4: Execution time in seconds for implementation vs problem size

| problem size | 2000 | 3000 | 4000 | 5000 |
|---|---|---|---|---|
| pboot1 | 4.86 | 5.66 | 5.01 | 5.56 |
| pboot2 | 8.58 | 10.85 | 10.99 | 12.04 |
| pboot3 | 7.05 | 7.84 | 7.86 | 9.33 |

Table A.5: Speedup with data size

## A.4 Replications

The figure 6.3 was created using the results of benchmarks executed on 16 CPUs with data set of 3000 integers listed in table A.6 and A.7.

| replications | 20000 | 30000 | 40000 | 50000 |
|---|---|---|---|---|
| boot | 47.8 | 79.2 | 103 | 119 |
| pboot1 | 9.7 | 14 | 19.4 | 21.2 |
| pboot2 | 5.5 | 7.3 | 8.6 | 10.4 |
| pboot3 | 6.8 | 10.1 | 11.2 | 13.2 |

Table A.6: Execution time in seconds for implementation vs number of replications

| replications | 20000 | 30000 | 40000 | 50000 |
|---|---|---|---|---|
| pboot1 | 4.93 | 5.66 | 5.31 | 5.61 |
| pboot2 | 8.69 | 10.85 | 11.98 | 11.44 |
| pboot3 | 7.03 | 7.84 | 9.20 | 9.02 |

Table A.7: Speedup with number of replications

## A.5 Complexity

The figure 6.4 was created using the results of benchmarks executed on 16 CPUs with data set of 3000 integers and 30000 replications listed in table A.8 and A.9.

| complexity | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| boot | 79.2 | 127 | 192 | 235 |
| pboot1 | 14 | 18.3 | 23.4 | 26.9 |
| pboot2 | 7.3 | 11.1 | 16.7 | 20.9 |
| pboot3 | 10.1 | 13.9 | 16.2 | 20.1 |

Table A.8: Execution time in seconds for implementation vs statistical function complexity

| complexity | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| pboot1 | 5.66 | 6.94 | 8.21 | 8.74 |
| pboot2 | 10.85 | 11.44 | 11.50 | 11.24 |
| pboot3 | 7.84 | 9.14 | 11.85 | 11.69 |

Table A.9: Speedup with complexity of statistical function

# Bibliography


[1] *What is R?* http://www.r-project.org/about.html

[2] *About Bioconductor* http://www.bioconductor.org/about/index.html

[3] Jon Hill, Matthew Hambley, Thorsten Forster, Muriel Mewissen, Terence M Sloan, Florian Scharinger, Arthur Trew, Peter Ghazal *SPRINT: A new parallel framework for R*

[4] Muriel Mewissen *SPRINT - User Requirements Survey Results* http://sprint.gti.ed.ac.uk/Docs/EXTERNA_SPRINT-URSR_v1.1.pdf

[5] Richard A. Becker *A Brief History of S* http://cm.bell-labs.com/stat/doc/94.11.ps

[6] *The Comprehensive R Archive Network* http://cran.r-project.org/

[7] *MPI: A Message-Passing Interface Standard* http://www.mpi-forum.org/docs/mpi21-report.pdf

[8] *PVM: Parallel Virtual Machine* http://www.etlib.org/pvm3/book/node17.html

[9] *Simple Parallel R INTerface* http://www.r-sprint.org/

[10] *Bootstrap and jackknife resampling in scientific knowledge advancement* Journal of Theory Construction and Testing, March 22 2002

[11] *Collins internet-linked dictionary of Statistics.* ISBN 0-00-720790-5

[12] Ader H. J., Mellenbergh G. J., Hand, *Advising on research methods: A consultant's companion* ISBN 9789079418015

[13] B.Efron. *Computers and the Theory of Statistics Thinking the Unthinkable.* SIAM Review, Vol. 21, No. 4, October 1979.

[14] Wolfgang Huber, Anja von Heydebreck, Martin Vingron *Analysis of microarray gene expression data* http://www.ebi.ac.uk/huber/docs/hvhv.pdf

[15] M. Kathleen Kerr and Gary A. Churchill *Bootstrapping cluster analysis: Assessing the reliability of conclusions from microarray experiments* Proc. Natl. Acad. Sci. USA, 98:8961. 8965, 2001.

[16] *EPCC - ness* http://www.epcc.ed.ac.uk/facilities/ness/

[17] *boot: Bootstrap R (S-Plus) Functions* http://cran.r-project.org/web/packages/boot/index.html

[18] *rlecuyer: R interface to RNG with multiple streams* http://cran.r-project.org/web/packages/rlecuyer/index.html

[19] *About Rmpi* http://www.stats.uwo.ca/faculty/yu/Rmpi/

[20] *MPICH-A Portable Implementation of MPI* http://www.mcs.anl.gov/research/projects/mpi/mpich1-old/

[21] *About MPICH2* http://www.mcs.anl.gov/research/projects/mpich2/about/index.php?s=about

[22] *LAM/MPI Overview* http://www.lam-mpi.org/about/overview/

[23] *Open MPI: Open Source High Performance Computing* http://www.open-mpi.org/

[24] *snow (an acronym for Simple Network Of Workstations)* http://www.sfu.ca/ sblay/R/snow.html

[25] *Writing R Extensions* http://cran.r-project.org/doc/manuals/R-exts.html

[26] John M. Chambers *Software for Data Analysis, Programming with R* pg. 422