INVESTIGATING PARALLELIZATION OF THE

ROBUST MULTI-ARRAY AVERAGE EXPRESSION

Spilios Alexopoulos

August 18, 2011

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2007
Abstract

Biostatistics refers to the application of statistical analysis on biological data. It can serve a variety of sciences including Biology and Medicine. However, the need to deal with large amounts of data can only be fulfilled by using all of the available computational power.

Most times, researchers do not have access to large computational infrastructures while in other cases they do not have the background to build efficient parallel applications. For both issues the SPRINT project aims to offer a solution. Taking under consideration that most modern computer systems are based on multicore architectures, parallelism can be a solution regardless of whether or not scientists have access to HPC machines.

SPRINT provides parallelised version of R functions while simultaneously maintaining an interface as similar as possible to the original sequential function. In an effort to prioritise the future functions to be added, the SPRINT team conducted an R user Requirement Survey. Based on the survey’s results, the RMA function is the subject of this thesis. This function calculates the Robust Multichip Average of a dataset.

The data described on a Microarray have to be transformed to an Expression Set clear from statistical noises before further calculations take place. This process is proved to be memory intensive. As noted by R users, apart from the performance issues, because of the large datasets and their limited resources, the function can be prevented from being executed entirely.

To achieve parallelisation we examined the two main parallelisation approaches supported by SPRINT; executing the already existing sequential application on each available processor on part of the data and creating a new parallel function. Using the sequential code was proven to be inappropriate for the RMA function since its execution required access to the entire dataset. Additionally, the performance improvement was not as satisfactory as expected. On the other hand, developing a new parallel function was harder to implement while its scalability heavily relies on the size of the data.

It was concluded that the parallelisation performance is limited by the sequential parts of the algorithm. For further improvement of the RMA’s performance more sequential fractions of the source code have to be parallelised.
# Table of Contents

List of Tables ......................................................... iii
List of Figures ......................................................... iv
List of Listings .......................................................... v
Acknowledgements ...................................................... vi

Chapter 1 - Introduction ............................................. 1
  1.1 Structure ....................................................... 2
  1.2 Conventions ................................................... 3

Chapter 2 – Background .............................................. 4
  2.1 The R Project ................................................... 4
    2.1.1 Garbage Collection ......................................... 4
    2.1.2 SEXP Type ................................................ 4
  2.2 Biostatistics concepts .......................................... 5
  2.3 The rma Function .............................................. 5
  2.4 The SPRINT Framework ....................................... 7
  2.5 The System .................................................... 7
  2.6 The Dataset .................................................. 7

Chapter 3 – The rma Function ...................................... 8
  3.1 rma Execution ................................................ 8
  3.2 rma Basics .................................................. 9
  3.3 rma Flow .................................................. 10
  3.4 Profiling the rma ........................................... 12
    3.4.1 Using the R Profiler ..................................... 12
    3.4.2 Adding Timers .......................................... 13
  3.5 Conclusions ................................................ 14

Chapter 4 – Using SPRINT ......................................... 15
  4.1 SPRINT Functionality ....................................... 15
  4.2 Adding Functions to SPRINT ............................... 16

Chapter 5 – Sprint Case-Studies .................................. 17
  5.1 Using the Sequential Function ............................. 17
  5.2 Implementing A Parallel Function ......................... 18
List of Tables

Table 3.1 – RMA Argument list ........................................ 9
Table 3.2 – Argument List, rma_c_complete ...................... 10
Table 3.3 – Argument List, rma_c_call .............................. 11
Table 3.4 – RMA Profile Output ..................................... 13
Table 3.5 – RMA Timer’s Results ................................. 14
Table 4.1 – SPRINT structure ...................................... 16
Table 7.1 – Parallel Background Correction Performance .... 29
Table 7.2 – Data Parallel Performance ............................ 33
Table 8.1 – Determine Target Performance ..................... 39
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Background Adjustment [9]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Quantile Normalisation [9]</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>rma_c_complete Flow-Chart</td>
<td>11</td>
</tr>
<tr>
<td>7.1</td>
<td>Background Correction Parallelisation Options</td>
<td>23</td>
</tr>
<tr>
<td>7.2</td>
<td>Column Consideration</td>
<td>26</td>
</tr>
<tr>
<td>7.3</td>
<td>Data Distribution on 4 Processors (P1, P2, P3, P4)</td>
<td>27</td>
</tr>
<tr>
<td>7.4</td>
<td>Data Reassembling</td>
<td>28</td>
</tr>
<tr>
<td>7.5</td>
<td>Background Correction (Data Parallel) Speedup</td>
<td>34</td>
</tr>
<tr>
<td>7.6</td>
<td>PRMA Speedup</td>
<td>34</td>
</tr>
<tr>
<td>8.1</td>
<td>Cyclical Data Distribution</td>
<td>38</td>
</tr>
<tr>
<td>8.3</td>
<td>Normalisation Speedup</td>
<td>40</td>
</tr>
<tr>
<td>9.1</td>
<td>Speedup Comparison</td>
<td>42</td>
</tr>
</tbody>
</table>
## List of Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>R Profiler Example</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>SPRINT’s execution script</td>
<td>15</td>
</tr>
<tr>
<td>6.1</td>
<td>rma to prma comparison</td>
<td>21</td>
</tr>
<tr>
<td>7.1</td>
<td>Data Dependence Example</td>
<td>24</td>
</tr>
<tr>
<td>7.2</td>
<td>Parallel Function Call</td>
<td>25</td>
</tr>
<tr>
<td>7.3</td>
<td>Background Correction Loop</td>
<td>26</td>
</tr>
<tr>
<td>7.4</td>
<td>Background Correction Call</td>
<td>32</td>
</tr>
<tr>
<td>8.1</td>
<td>Calling rma_c_call</td>
<td>37</td>
</tr>
<tr>
<td>8.2</td>
<td>Normalisation Call</td>
<td>37</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank my supervisor, Mr Terry Sloan, for his guidance and enthusiasm throughout the whole project. I also would like to thank Michal Piotrowski and Lawrence Mitchell for sharing their knowledge and experience of SPRINT.

In addition, I would like to acknowledge Dr Donald Dunbar for his kindness in providing me with experimental datasets as well as helping me establish the necessary biostatistical background for my research.
Chapter 1 - Introduction

The need for computational power and large amounts of memory in biostatistics rises as the need for more accurate computations increases. The DNA data are represented as DNA microarrays. These microarrays can be up to several Gigabytes in size. Taking under consideration that some operations require the entire dataset to be loaded on the memory, one can imagine how this creates problems for the scientists. In the genomic data analysis, memory is the most important limiting factor. Furthermore, it is also widely understood that having to deal with large datasets requires as much computational power as can be attained.

Unfortunately, most of the scientists do not have access to HPC machines. However, multicore machines are now dominating the market. It would be beneficial if scientists could take full advantage of any infrastructure that they have access to. In either case, multiprocessor programming requires specialised skills and more effort and time than biostatisticians would be able to sacrifice from their actual research.

For these reasons EPCC in collaboration with the Division of Pathway Medicine at the University of Edinburgh have developed the SPRINT framework. SPRINT is developed on top of R which is one of the most widely used statistical frameworks. Its name is an acronym of Simple Parallel R INTerface and it aims to allow scientists access to parallelised versions of the R functions that they already use, keeping the function’s interface as close as possible to the original function.

Parallelisation can either lead to faster execution or it can enable functions that were not able to be executed on systems with limited resources, to run. As the genomic data research is highly connected with disease treatment, it is obvious how important it can be to enable researchers to use any available means, including High Performance Computing.

In this project we will focus on the parallelisation of the Robust Multichip Average expression and how this can be achieved. This procedure includes operations that ‘prepare’ the dataset before any calculations take place. The term preparation is used to refer to the corrections that need to be made at the dataset, due to statistical errors.
1.1 Structure

This dissertation is divided in 9 chapters and its structure is presented below:

Chapter 2 includes all the background information that we consider important for the reader to know before proceeding to the main focus of this paper. Content from several different scientific fields is included within the scope of our background information. Initially the R project and its basic concepts are presented. Afterwards, some basic biostatistics terminology is explained before the RMA functionality is described. Then a quick reference to the SPRINT framework is made and finally the system that we use is presented.

In the 3rd chapter we describe in depth the functionality of the RMA function. In addition, we examine the sequential R function and make a first attempt to identify the overhead of its execution.

The 4th chapter focuses on the SPRINT framework’s structure. Within this chapter it is explained how a new function can be added into SPRINT and what the standard procedure that has to be followed is.

Before analysing the parallelisation options of the RMA function itself, we considered it useful to dedicate the 5th chapter to presenting some previous case studies of other functions that have been parallelised within SPRINT. Our intention was to include functions that had different approaches for achieving parallelisation.

Eventually, in the 6th chapter the work that had to be done before starting the actual parallelisation is described. This includes description of the SPRINT’s configuration files updating procedure and the implementation of a test application that was used to validate our results.

Chapter 7 demonstrates how parallelisation can be achieved on the Background Correction function and describes the different approaches that we implemented. It concludes with a performance comparison among these implementations.

The 8th chapter focuses on the Normalisation function and its parallelisation. The decomposition strategy that was followed is compared with the previous cases that were examined. A performance analysis also takes place.

In the final chapter all previous conclusions have been unified into a comprehensive analysis. Additionally, a review of the problems encountered in consideration of our initial risk analysis is discussed. Finally, some suggestions for future research are given.
1.2 Conventions

The following conventions are used in this thesis:

$  

The dollar symbol ($) is used to indicate commands of the LINUX command prompt.

>  

The greater symbol (>) indicates commands under the R statistical environment.
Chapter 2 – Background

In this chapter we will present the background concepts pertinent to our research. These include the tools that have been used (both software and hardware) and some basic information about the use and the need for parallelisation of the Robust Multichip Average procedure.

2.1 The R Project
R is a scripting language and an environment used for statistical analysis. The R environment is based on a former platform called ‘S’ and was developed at Bell laboratories. It is part of the GNU project which means that it is an open source project and compatible with most of the UNIX based systems. R is also available on Windows and MacOS. R’s greatest advantage is its high extensibility through different packages [1].

In our research we will focus on R’s usage in biostatistics. Because of its advantages that we have already mentioned (powerful, extensible, non-commercial) R is one of the most important tools in biostatistics [2].

Some of the R concepts that will be useful to know for the extent of this paper are listed below:

2.1.1 Garbage Collection
The R users are not responsible for freeing the memory allocated for R objects. On the contrary, a mechanism called ‘Garbage Collector’ periodically clears the memory which is not used.

In case that an R object is created within C code, it is the developer’s responsibility to ensure that this will not be destroyed by the garbage collector. This is achieved with the use of the ‘PROTECT’ macro that actually informs R that an object is being used.

Respectively, the ‘UNPROTECT’ macro indicates that an object is not in use anymore. The protection mechanism ‘is stack-based, so UNPROTECT(n) unprotects the last n objects which were protected. The calls to PROTECT and UNPROTECT must balance when the user’s code returns’ [3].

2.1.2 SEXP Type
“What R users think of as VARIABLES or OBJECTS are symbols which are bound to a value” [3]. During this project, these values were mostly SEXP pointers. R provides an interface to
compiled C code, through the `.Call` function [4]. The arguments passed are intercepted by C as an array of SEXP types.

In order to use these objects they have to be ‘sanitised’ through the ‘NUMERIC_POINTER’ function which removes the R headers of the object and transforms it to a regular C pointer.

### 2.2 Biostatistics concepts

Since this paper is not targeted towards people with biological or biostatistician background, it is necessary that some basic biostatistics concepts and terminology be explained. However, we will try to include enough information in the interest of explaining what the RMA function is used for. Hopefully, the better understanding of the function will lead to discovering the potential parallelisation opportunities.

A DNA MicroArray (also called chip) is considered as a grid of DNA spots on a substrate, called probes, and it is used to detect complementary sequences. Biostatisticians usually use DNA microarrays to ‘measure the expression levels of large numbers of genes simultaneously or to genotype multiple regions of a genome’ [5]. The expression arrays are mostly used to measure the mRNA levels in a cell or the average mRNA levels in a population of cells.

A Microarray includes several ‘probesets’. ‘These probesets are intended to measure expression for a specific mRNA. Each probe is complementary to a target sequence which is derived from one or more mRNA sequences. probesets consist of 25mer probe pairs selected from the target sequence: one Perfect Match (PM) and one Mismatch (MM) for each chosen target position’ [6].

The most common file type to represent the MicroArray information is a CEL file. The CEL file stores the results of the intensity calculations on the pixel values of the DAT file. A single representative intensity value is stored per cell (feature) of the image [7]. Test datasets of CEL files are provided by Affymetrix' GeneChip® technology. Each of the CEL files within the same experiment has the same number of probes regardless. In terms of our project we can assume that a probe is a part of the dataset represented within the input files.

For each chip there also exists a CFD File (Chip Description File) to describe ‘probe locations and probeset groupings’ [8].

### 2.3 The rma Function

As we have described, there is a great number of intensity values within each probe and many probes are grouped together into probesets. There are several algorithms to transform these intensity values to expression sets. Robust Multichip Average (RMA) is one of these.
During our research we will focus on the RMA function which is part of the 'Bioconductor affy package'. Bioconductor is an open source project that provides tools for the genomic data analysis which is mostly based on R. The 'affy package' contains functions for exploratory oligonucleotide array analysis [9].

The 'affy RMA function' computes 'the Robust Multichip Average expression measure described in Irizarry et al Biostatistics (2003)' [9]. It consists of three steps: a background adjustment, quantile normalization and summarization.

The background adjustment is used to remove the noise from the signal [Figure 2.1].

![Background Adjustment](image1)

**Figure 2.1 – The Background Adjustment [10]**

Normalization is a process of reducing unwanted variation across chips that is created due to non-biological factors. After normalization the quantiles should be the same across all the chips [Figure 2.2]. The term 'quantile' in statistics describes a subsection of a frequency distribution. The quantiles within a single distribution have to be the same size.

![Quantile Normalisation](image2)

**Figure 2.2 – Quantile Normalisation [10]**

Finally, summarization is the reduction of ‘the 11-20 probe intensities for each probeset on to a gene expression value’ [9].
2.4 The SPRINT Framework

SPRINT is a parallel interface for R developed by EPCC in collaboration with Division of Pathway Medicine. In order to simplify SPRINT’s concept we could say that it is an R package that consists of parallelised biostatistics R functions. SPRINT requires R to be installed, but unlike R, it is only available on LINUX based platforms.

SPRINT uses C and MPI. Consequently, an implementation of MPI-2 has to be installed in order to install and use SPRINT [11]. Apart from that, SPRINT is installed and loaded as a regular R library.

2.5 The System

Most of the software development that has been done for this project, took place on a dual core laptop under UBUNTU OS. Both R and SPRINT are available in most platforms.

However, in order to have more accurate timings and test our code on more processors, NESS was also used. NESS is an SMP cluster provided by EPCC mostly to ‘support local projects and some general research activities’ [12]. It is based on 2.6 GHz AMD Opteron processors.

NESS uses the scientific distribution of Linux. The Sun Grid Engine (SGE) is used for job submission to the 32 processors of NESS’s backend.

The MPI library is available on NESS. By default the MPI library is based on the PGI compilers. However, in order to install SPRINT, as noted by Dr Lawrence Mitchell, we have to switch to the shared version of the mpich2 module:

$ module swap mpich2/pgi mpich2/1.0.7p4-ch3_sock-gcc4-shared

2.6 The Dataset

The dataset used for timings throughout this project is provided by Affymetrix. Affymetrix is a company built on top of the GeneChip® technology which allows storing large amounts of biological data on a small glass chip. On their web site they offer a variety of test datasets [13].

The selected dataset’s size is more than 700MB but this is comparatively small to the other test data. However, in terms of this research it is adequate to give rational execution times.
Chapter 3 – The rma Function

In this chapter we will present how the RMA algorithm works and how it is implemented within the R framework. We will focus on the programming aspect of the function rather than the biostatistical. We will also examine how the data is treated during the execution. Furthermore, we will measure its performance and try to locate the main bottleneck of the procedure.

The main aim of this chapter is to establish the necessary background before considering the parallelisation opportunities of the RMA function.

3.1 rma Execution

Before executing the RMA function the ‘affy package’ has to be installed and loaded into R. Afterwards, the RMA’s input dataset has to be read and stored into an R object. This can be achieved with the ReadAffy() function which is also included in the ‘affy package’. Specifically, if the ReadAffy() function is called without any arguments, all the CEL files is outlined in ‘Table 3.1’.

It is important to note that the argument list has remained the same as previous RMA versions, most likely in favour of keeping the same interface. Specifically, the destructive argument as commented in the RMA’s source code does not have to be changed from its default value in the current RMA version. Furthermore, the ‘bgversion’ argument is not used at all within the code. Finally, after conferring with Dr Donald Dunbar, it was established that there is not any obvious reason as to why the RMA execution should not include either background correction or normalisation. In practice, this renders both ‘normalize’ and ‘background’ arguments superfluous.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>The affybatch object that was read with the ReadAffy() function</td>
</tr>
<tr>
<td>subset</td>
<td>‘a character vector with the the names of the probesets to be used in expression calculation’ [9]</td>
</tr>
<tr>
<td>verbose</td>
<td>Boolean variable that determines whether the program will print information messages or not</td>
</tr>
<tr>
<td>destructive</td>
<td>Boolean variable that determines whether the rma function will work on the object or on a copy of it.</td>
</tr>
<tr>
<td>normalize</td>
<td>Boolean variable that determines whether the normalisation function will be executed or not</td>
</tr>
<tr>
<td>background</td>
<td>Boolean variable that determines whether the background correction function will be executed or not</td>
</tr>
<tr>
<td>bgversion</td>
<td>Determines the background correction function’s version that will be used</td>
</tr>
</tbody>
</table>

Table 3.1 – RMA Argument list

3.2 rma Basics

As we have already mentioned there are three major steps during the execution of the function. These are the background correction, a quintile normalisation and finally the summarisation. In this chapter we will have a closer look at how these are implemented in the RMA function.

Background correction refers to the operations applied to the data in order to remove non-biological contributions (statistical noises) in the measured signal [14].

Normalization “refers to computational data transformations intended to remove certain systematic biases from microarray data, such as dye effects, intensity dependence, and spatial or print-tip effects” [14].

In algorithmic terms Normalization is a four step procedure, assuming an array of genomic data as its input:

1. Sorts the columns of the array
2. Calculate the average of each row
3. Replace all elements of each row with the calculated average
4. Return each element into its initial position

Concluding this section, we would like to clarify that the reason to execute the RMA function is to make all the samples/chips comparable.
3.3 rma Flow
The aim of this section is to investigate and present RMA’s functionality. We will try to identify any internal function calls that take place throughout RMA’s execution. Additionally, it would be interesting to determine if and how the arguments can affect the flow of the function.

At first, the R rma() function depending on the ‘destructive’ argument, calls either the ‘rma_c_complete’ or the ‘rma_c_complete_copy’ function. These two functions have the same functionality with the only difference to lie on the fact that the ‘rma_c_complete_copy’ creates a copy of the dataset. As it is mentioned in the previous section there is not any reason to use the ‘rma_c_complete_copy’, therefore for the rest of the paper, we will focus on ‘rma_c_complete’.

The 'rma_c_complete' function is a C function. We will examine the data flow in order to identify any data dependencies and eventually uncover any potential decomposition strategies.

The arguments passed from the R function to the 'rma_c_complete' are presented in ‘Table 3.2’. All these arguments are passed as SEXP pointers.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pm(object,subset)</td>
<td>By default it is all the object, if subset is given it passes only the Perfect Matches</td>
</tr>
<tr>
<td>pNList</td>
<td>This is the result of the probeNames(object,subset). The ‘probeNames function’ returns the probe names associated with the rownames of the intensity matrices one gets with the ‘pm’ and ‘mm’ methods.</td>
</tr>
<tr>
<td>ngenes</td>
<td>This is the result of ‘length(geneNames(object))’, although the ‘ngenesis’ is supposed to be deprecated and it is replaced by ‘featureNames’. Both functions return the probe set names also referred to as the Affymetrix IDs.</td>
</tr>
<tr>
<td>normalize</td>
<td>Same as the RMA argument</td>
</tr>
<tr>
<td>background</td>
<td>Same as the RMA argument</td>
</tr>
<tr>
<td>bgversion</td>
<td>Same as the RMA argument</td>
</tr>
<tr>
<td>verbose</td>
<td>Same as the RMA argument</td>
</tr>
</tbody>
</table>

Table 3.2 –Argument List, rma_c_complete

If the background argument is set to TRUE, the rma_bg_correct function is called. The rma_bg_correct function is part of the 'Bioconductor preprocessCore'. The preprocessCore
is an R package that includes low-level pre-processing routines and is required and imported by the affy package.

The arguments passed in this function are: a pointer to the data object (in case that the subset is false this is the whole object described by the CEL files), the number of rows and the number of columns of the object as given by the ‘getAttrib’ function. The ‘getAttrib’ function gets individual attributes for an object depending on the second argument which in this case is ‘R_DimSymbol’. The rma_bg_correct function does not return anything. However, it applies the background correction operations on the input data object.

Subsequently, the ‘rma_c_call’ function is called and its output is what is finally returned to R. The ‘rma_c_call’ function is called with the background corrected object and the arguments from the rma_c_complete function that will now be needed are presented in Table 3.3. Included in this function are both the normalisation and the expression set calculation operations. The normalisation is implemented by the qnorm_c function, which is also part of the preprocessCore package. Finally, the normalised dataset is used for calculating the Expression Set.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pmmat</td>
<td>The object after background correction</td>
</tr>
<tr>
<td>ProbeNamesVec</td>
<td>The pNList argument from the ‘rma_c_complete’ call</td>
</tr>
<tr>
<td>N_probes</td>
<td>The ngenes argument from the ‘rma_c_complete’ call</td>
</tr>
<tr>
<td>norm_flag attribute</td>
<td>The normalize argument from the ‘rma_c_complete’ call</td>
</tr>
<tr>
<td>verbose</td>
<td>The verbose argument from the ‘rma_c_complete’ call</td>
</tr>
</tbody>
</table>

Table 3.3 – Argument List, rma_c_call

Eventually, the object is returned back to the R function, before the fulfilment of the RMA function’s execution.

![Figure 3.1 – rma_c_complete Flow-Chart](image-url)
3.4 Profiling the rma

Profiling is the first step of performance optimisation. It allows the software developer to identify the causes of overhead. The R framework has its own profiling tool. In this chapter, we will use the R profiling tools and more profiling techniques will be applied, if necessary.

3.4.1 Using the R Profiler

The ‘Rprof’ command enables the R profiler while the same command with 'NULL' passed as argument disables it. An example of the potential use follows below:

```r
> Rprof('rma.out')
> rma(data)
> Rprof(NULL)
```

Listing 3.1 – R Profiler Example

The profiler produces an output file whose name has been defined as an input argument in the Rprof command. The output can be viewed with the following command:

```
$ R CMD Rprof rma.out
```

In order to have more accurate and reliable profiling results in our experiment, we used the test dataset provided by ‘Affymetrix’ in order to increase the execution time. The execution of the RMA function took place on NESS. The most important calls in terms of their execution time are demonstrated in the following table while all the output results are presented in Appendix A. The information included in the table consists of:

- %self: The percentage of the total execution time that was spent on this function without including interior calls to other functions.
- Seconds self: The absolute execution time in seconds that was spent on this function without including interior calls to other functions.
- %total: The percentage of the total execution time that was spent on this function including interior calls to other functions.
- Seconds total: The absolute execution time in seconds that was spent on this function including interior calls to other functions.
It is noticeable that the execution of the RMA is dominated (more than 85% of the total execution time) by the .Call function. The .Call function is used to make calls to compiled code that has been loaded into R [4]. That indicates our research should focus on the C functions called within the RMA function instead of the R part of the function.

However, the standard R profiler can only profile the R functions. Consequently, other profiling tools have to be used to determine the causes of the overhead within the C code. Unfortunately, the standard C profiles do not seem to be applicable. When the C code is compiled as part of an R package, instead of an executable file, an object is created. This object cannot be profiled neither by ‘gprof’ or ‘prof’ tools.

### 3.4.2 Adding Timers

Since the standard R profiler cannot provide satisfactory information for our research and the standard C profilers cannot be applied at all, timing specific fractions in the C code will have to be used in their stead to achieve our aims.

In order to use the standard C time functions we have to include the 'time header library'. The ‘time.h’ is a header file defined in the C Standard Library to declare time and date functions that provide standardized access to time/date manipulation and formatting [15]. In order to perform the timings we will have to set two variables t1 and t2 of ‘time_t’ type. This data type is used for storing system time values. In order to time a specific part of our code we will have to initialise the t1 timer before and the t2 timer after its execution. By subtracting these two values we will have the exact time taken for the execution of that fraction.

The important decision that has to be made is what parts of the code should be timed. Based on our findings by examining the code, we will time the background correction function (rma_bg_correct), the normalisation function (qnorm_c) and the fraction of the code that calculates the Expression Set.

The RMA function was executed on our test dataset on NESS and the values recorded is a representative sample taken after several executions. In a total execution time of 92 seconds, the background correction lasts 43 and the normalisation 20. On the other hand

<table>
<thead>
<tr>
<th></th>
<th>% self</th>
<th>seconds self</th>
<th>% total</th>
<th>seconds total</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Call</td>
<td>85.4</td>
<td>80.00</td>
<td>85.4</td>
<td>80.00</td>
</tr>
<tr>
<td>.local</td>
<td>6.7</td>
<td>6.24</td>
<td>13.6</td>
<td>12.74</td>
</tr>
<tr>
<td>ls</td>
<td>1.7</td>
<td>1.56</td>
<td>1.7</td>
<td>1.56</td>
</tr>
<tr>
<td>load</td>
<td>1.5</td>
<td>1.44</td>
<td>1.5</td>
<td>1.44</td>
</tr>
<tr>
<td>as.vector</td>
<td>1.4</td>
<td>1.32</td>
<td>1.4</td>
<td>1.32</td>
</tr>
</tbody>
</table>

Table 3.4 – RMA Profile Output
calculating the expression set is proven to be insignificant. The rest of the time is spent in other less important R function calls. It was discovered that the results are in fact in accordance with the R profiling that we have done.

<table>
<thead>
<tr>
<th></th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background correcting</td>
<td>43</td>
</tr>
<tr>
<td>Normalizing</td>
<td>20</td>
</tr>
<tr>
<td>Calculating Expression</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.5 – RMA Timer’s Results

3.5 Conclusions
To summarise, in this chapter we have described in detail the flow of the RMA function. The transition from R to the C functions has also been described. In addition, a first impression of the data flow has also been made.

However, the most important finding is in regards to the profiling that we have already completed. It is now obvious that the main overhead of the RMA function is created at the execution of the rma_bg_correct and the q_norm functions. Both of them are part of the preprocessCore package. The preprocessCore package is also developed by BioConductor and is installed automatically as a requirement of the affy package.

Eventually, these two functions will be the starting point for our parallelisation efforts.
Chapter 4 – Using SPRINT

This chapter intends to present SPRINT from the developer’s point of view. We will therefore present how a function could be added in the framework. Additionally the SPRINT’s functionality will be analysed, in terms of what exactly takes place in the background when calling a SPRINT function.

4.1 SPRINT Functionality
In order to execute a SPRINT function in parallel we have to execute R in parallel as well. Thus, we have to write a script where the SPRINT library is loaded and then execute the parallel function. As presented in the ‘SPRINT user guide’ [11] a simple script to execute the included SPRINT test function is:

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

Listing 4.1 – SPRINT’s execution script

Eventually, this script –assuming that we call it ‘test.R’- has to be executed via MPI:

```
$ mpiexec -n 4 R -f test.R
```

This command executes R on four processors and executes the ‘test.R’ script on all of them. All the processors load the SPRINT library and execute the same function (in this case the ptest() function). However, all the processors apart from the master are in a ‘wait mode’ until they are awakened by the master process.

The master process starts the execution of an R-stub and when it meets a SPRINT function that has to be executed in parallel, sends an MPI command. This MPI message contains a unique code for each of the SPRINT functions and wakes up the worker processes. The worker processes execute their function and when finished go back to the ‘wait mode’.

Except for the function code that is passed from the master process to the workers any other communication (e.g. data distribution, data collection, halo swapping) is done via MPI as well.
4.2 Adding Functions to SPRINT

The SPRINT package has a specific structure as all the functions included have to share the same interface. When adding a function the R code and the C source code both have to be under specific directories as shown below. In addition some configuration files have to be updated.

SPRINT is organised into the following directory structure:

```
/ -- root
|-- man
|-- our_macros
|-- R
|-- src
```

Table 4.1 – SPRINT structure

Like any other R package, SPRINT includes a ‘man’ directory. For each function there has to be a help page accessible through R. This includes information about the function, most times including a general description of the function, examples of usage and information about the authors.

The ‘our_macros’ directory consists of general purpose macro instruction.

The ‘R’ directory includes the R function that is called from the R framework. In SPRINT, it is mostly used to read the input data, check their correctness, call the function where the parallelisation takes place and eventually return the result to the R framework.

The ‘src’ directory includes subdirectories for each of the SPRINT functions with the name of each function respectively. In any of these, there is the interface and the implementation subdirectory. The ‘interface’ subdirectory includes the source code for the function that is called by the R function that was previously mentioned. As it can be assumed this function implements the basic SPRINT functionality and works as a link between R and SPRINT. Finally, the ‘implementation’ subdirectory includes all the functionality of the parallelised code.

In addition to all the previous, the following configuration files have to be updated when adding a new SPRINT function. Firstly, in the NAMESPACE file a record of the new function has to be added in order to enable R to find the new function. As it was previously mentioned when the master process wakes the workers up, an MPI message containing an enumeration code is sent. Hence, the new code has to be added in the ‘commandCodes’ enumeration field which is located in the ‘functions.h’ header file.
Chapter 5 – Sprint Case-Studies

In this chapter we will examine the different approaches that we can apply on parallelisation of the RMA function within SPRINT.

Before describing the main approaches supported by the SPRINT architecture, it would be useful to make a reference to the two basic parallelisation models; the data parallel and the task parallel. Data parallelism means that each processor executes the same calculations on different parts of distributed data. In most cases there is a master process that controls the data distribution. On the other hand task parallelism is achieved by assigning on each process an independent task on the same or different data.

The first parallelisation option is to use the already existing R (or C) function and execute it on each available process either on different dataset (data parallel model) or on the same dataset by combining the results returned by each of them (task parallel model). This approach is easy to implement but it is algorithm specific and cannot always be applied.

The other option is to implement a parallel version of the function that we are interested in. Even though this approach requires much effort, it gives more flexibility to the developer. Flexibility here encompasses the ability to use several decomposition strategies as well as build a new and more efficient parallel algorithm.

In order to understand better how these models can be applied with SPRINT, some case studies of already parallelised R functions will be presented. These case studies consist of:

- the Random Forest classification function
- the Pearson’s pair-wise correlation function

5.1 Using the Sequential Function

A case study that implements the model of using the already existing sequential algorithm is the parallelisation of the Random Forest classification.

Tree learning is a method that is used to construct predictive models which map data observations to some conclusions about the data. ‘The random forest algorithm is an ensemble tree classifier that constructs a forest of classification trees from bootstrap samples of a dataset’. [16]
In terms of its parallelisation, data parallel approaches do not map well onto microarray data in this algorithm. Thus a task parallel approach is preferred. The steps followed in that implementation are:

a. The trees are equally divided between available processes.
b. The ‘subforests’ are generated in parallel by the different processes.
c. A larger forest which is the combination of the returned subforests is created on the master process.

The Random Forest paradigm is a case of task parallel computations that use the already existing sequential code. Even though the idea of using the sequential code fits better with the data parallel model, this specific algorithm allowed the SPRINT team to apply this approach. This parallelisation effort, when executed on HECTOR\(^1\), led to speedup around 40 for typical use cases [16].

5.2 Implementing A Parallel Function
Even though it is harder to design a new parallel application, in most cases it is the only parallelisation option available. Most of the functions currently included in SPRINT, implement this approach. In terms of our research the parallelisation of the Pearson’s pairwise correlation will be presented.

The parallelisation approach can be described through the following steps [17]:

a) The master process distributes the data to the available worker processes
b) Each worker is assigned with one row of data at a time. The computation operations that take place on each row of data lead in storing the partial results into a local buffer.
c) When the computations on a row are finished the local buffer is sent to the master process and the worker is assigned with a new row. When all the rows are operated the master process terminates the workers.
d) The output procedure is executed sequentially on the master process. Since all the local buffers are gathered to the master process an output file is generated with the use of the write.table R function.

This parallel algorithm proved to have a serious bottleneck since each processor had to send its individual result to the master process. Another version that was designed to take advantage of parallel I/O was also implemented. The decomposition model remains the same but the local buffers are not returned to the master process after each row computation. Instead, all the processes write in parallel a file at the end of the program.

\(^1\)HECTOR is the UK National Supercomputing facility
Apart from achieving better performance due to taking advantage of parallel IOs, scalability is increased significantly.
Chapter 6 – Preparing For Parallelisation

In this chapter, we are going to describe all the necessary work that had to be done before the implementation of the RMA’s parallelisation begins. All the work that had to be done in order to submit the parallel RMA function to SPRINT is presented. Additionally, the design of the result verification mechanism that we use is also demonstrated.

6.1 Preparing SPRINT for rma

In chapter four, we have reviewed how a function can be added in SPRINT. Now we will apply this knowledge aiming to add the rma function to SPRINT. Before adding any code to SPRINT it is necessary that we update the configuration files so the SPRINT is aware of the new function.

First file to be updated is the NAMESPACE, located in the SPRINT root directory. A reference to the new prma function is made (Appendix B). Now R can locate the new prma function.

Furthermore, the command code that wakes up the worker processes and corresponds to the prma function has to be added. The enumeration list in the function.h header file has also to be updated with the new command code, PRMA (Appendix B).

Finally the function that is called by the ‘interface’ function has to be declared as external. Additionally, the look-up table that ties up the function pointers and the commandCode enumeration has to be updated. All the above updates take place within the src/algorithms/common/functions.c file.

6.2 Adding rma into SPRINT

Since the SPRINT configuration files are updated, we can proceed to the creation of the prma’s interface function. One of the fundamental requirements of the SPRINT project is to keep the parallelised function’s interface as similar to the sequential. The initial design of the sequential RMA function enables us to adhere to the aforementioned requirement.

It is worthwhile to mention that in order to keep the prma’s interface identical to RMA we have left the argument list as it was even though the new function does not need all the arguments. The destructive argument could be omitted as in the new version it is not at all needed. Even in the RMA its existence does not have any reason since it was only useful in prior versions.
The RMA function reads as an argument the dataset object, makes some operations on this object (keeps the Perfect Matches) and passes it along with some more input information to a C function. The result of the function is then returned. In other words, the R part of the RMA function does exactly what we would expect from an SPRINT-R interface function to do. Hence, the only thing that we have to change in order to have our R stub ready for SPRINT is to call another C function instead of the initial one. In the ‘Listing 6.1’ below, we isolated the part of the code that has been changed.

### RMA R function

```
... pNList <- probeNames(object,subset)
pNList <- split(0:(length(pNList) -1), pNList)

if (destructive){
    exprs <-
        .Call("rma_c_complete", pm(object,subset), pNList, ngenes, normalize, background, verbose, PACKAGE="affy")
} else {
    exprs <-
        .Call("rma_c_complete_copy", pm(object,subset), pNList, ngenes, normalize, background, bgversion, verbose, PACKAGE="affy")
}
... 
```

### PRMA R stub

```
... pNList <- probeNames(object,subset)
pNList <- split(0:(length(pNList) -1), pNList)

#destructive argument's functionality removed from prma function
exprs <-
    .Call("prma", pm(object,subset), pNList, ngenes, normalize, background, bgversion, verbose, PACKAGE="sprint")
... 
```

**Listing 6.1 – rma to prma comparison**

The prma C function that is called within the R stub is actually the interface function of SPRINT. In this part of code MPI is initialised and the worker processes are awakened by the master that is the only one to previously execute the code. This is relatively a standard procedure as described in previous chapter. Now the arguments are passed to the ‘rma_c’ function which replaces the sequential ‘rma_c_complete’ function and is included in the implementation directory of SPRINT. It is important to note that only the master process calls the function with the argument list. All the other processes do not have access to the data. Consequently, they call the same function without any arguments.

At this point it is ensured that all the available processes will call the function that is the subject of the parallelisation. The data that will be needed are available on the master process. During the implementation of the parallel code the worker processes will have to be sent the appropriate data for their operations.
The potential parallelisation approaches and the decomposition strategies will be discussed extensively in the proceeding content.

### 6.3 Results Validation

In order to validate our parallel code the Expression Set given by the sequential execution of the RMA function was compared to the Expression Sets that were returned by the prma. In the affy package the ‘write.exps(eset)’ method can be used to export an expression set to a text file. We can then compare the two text files. In order to have correct results, the elements have to be the same one-by-one.

However, it would be impossible to examine all the elements one-by-one manually. One option could be to use the UNIX ‘diff command’. Diff is used to find the differences between two files. During the development of the project, the need for a more flexible testing platform arose. Eventually, a simple application was built. The application reads and compares individually the elements of the two output files. The main difference, in terms of its functionality, to the diff command is that we can set a value of acceptable error inflation. This is useful to cover cases where there are small losses of precision due to message passing.

#### 6.3.1 Testing Application Implementation

As mentioned, a completely individual application was designed and implemented using the C language for testing and result evaluation purposes.

The application expects two files in its argument list. In case that the number of arguments is wrong it returns an error and terminates its execution. Since the number of arguments is correct the two files are opened only for reading. In case the files are not successfully opened the program exits.

The first line of the files is skipped since it contains the titles of the columns instead of numerical values. Eventually, a looping procedure throughout all the lines of the files starts. The elements of each line are stored in two temporary arrays, one for each file. The maximum size of the elements of the line (and the array) is defined via a compiler directive. Before another line is read the elements of the arrays are compared one-by-one. If their difference is bigger than the maximum error value, which is also defined as a compiler directive, then an error message is printed along with the line, the column and the values of the two elements. Additionally, in case the number of columns in the two files is not the same, it means that the input datasets were different and the comparison does not have any practical applications. Subsequently, in the aforementioned case the program also stops its execution and returns an error message.
Chapter 7 – Parallelising Background Correction

In this chapter, we will examine the background correction parallelisation options. During the previous timings that we have made, it is proved that the main overhead in the execution time of the RMA function is the background Correction function.

Since we are working within SPRINT we have two different approaches that we could follow. Either implementing the parallelisation on the background correction function itself or distributing the data to the worker processes and then calling the sequential background correction function for each of them (Figure 7.1).

7.1 Considering Background Correction Parallelisation

The first parallelisation option is to parallelise the rma_bg_correct function which is within the preprocessCore package. In that case the main advantage is the creation of a parallel function that would be potentially used by other functions apart from the RMA.

On the other hand, a data parallel approach would be much easier to implement. In addition, as proved by previous case studies a data parallel approach can result in good
performance. The main concept of data parallelism is that a large dataset is divided into smaller chunks that are distributed among the available processors. The processors operate these data in parallel and when they finish they send their data back to the master process where it is reassembled into a single data set. In order to achieve data parallelism in this project, each processor has to be assigned with a subset of the data and the ‘rma_bg_correct’ function has to be called by each processor separately. Data parallelism is one of the most popular techniques for multicore programming. However the main limitation is that all the data chunks have to be independent from each other. Of course this is algorithm specific, and not every problem can be solved this way.

According to the description of the background correction algorithm in Chapter 3, it seems that a data parallel approach could be implemented. However, before continuing the source code should be examined in order to identify any dependencies in case that the dataset is divided.

Even though the first impression is that a data parallel approach is applicable, this is not true. Within the code the use of the whole dataset for parameter calculation is needed. An example is given at the following fraction of code (Listing 7.1). The parameter ‘PMmax’ is calculated according to the entire data object. As shown in the example below the use of an altered ‘PMmax’ value in proceeding conditions will lead to different execution flow.

```c
PMmax = max_density(PM, rows, cols, column);
for (i=0; i < rows; i++){
    if (PM[column*rows + i] < PMmax){
        tmp_less[n_less] = PM[column*rows + i];
        n_less++;
    }
}
```

Listing 7.1 – Data Dependence Example

Consequently, a data parallel approach would result in a ‘locally’ background corrected version of the initial dataset. From a statistical scope this solution could be affordable, but in terms of the SPRINT project it is important to maintain results that are as close as possible to the sequential function.

As a result a parallel version of the background correction function will have to be implemented.
7.2 Background Correction Parallel implementation

In this case a parallel version of the ‘rma_bg_correct’ has to be developed. All the parallelisation operations will take place within this function, called ‘prma_bg_correct’. All the processes have to call the function but only the master process will pass the argument list.

7.2.1 Considering Parallelisation

As mentioned before, in this parallelisation approach, all the processes will have to call the ‘prma_bg_correct’ function. In ‘Listing 7.2’ we can see how the correct argument passing was achieved and what has to be done before implementing the ‘prma_bg_correct’ function. It is noticeable that the master process passes an argument list while all the other processes call the same function without arguments. At the time that the call happens, the worker processes do not have any access to the input data.

```c
if (worldRank == 0) {
    prma_bg_correct(0,PM, rows, cols);
} else {
    prma_bg_correct(0);
}
```

Listing 7.2 – Parallel Function Call

7.2.2 Applying Parallelisation

The original source code of this function included some POSIX thread parallelisation which proves that there was prior need for faster execution. Compiler directives determine whether the POSIX-threads will be used or not. POSIX is a shared memory programming model.

However, POSIX-threads are neither supported by SPRINT nor by distributed memory systems. Additionally, the POSIX-threads approach is fundamentally different than the Message Passing that we want to enact. Consequently, any POSIX parallelisation has to be removed in order to start from a pure sequential version. After removing any POSIX parallelisation, the code consists of two function calls within a loop statement (Listing 7.3). For a better understanding of the source code and eventually its parallelisation these functions were inlined.
The ‘rma_bg_parameters’ function endeavours to calculate three parameters. These parameters are used in the background adjustment operations that are implemented through the ‘rma_bg_adjust’ function. In a total execution time of 42 seconds for the background correction operations, 8.2 seconds were spent on the background adjustment process while the rest were spent calculating the parameters.

As it is presented previously (Listing 7.3), both functions are executed within a loop. Each call of the function refers to a column of the dataset. It is of vital importance to notice that what is considered to be a column is not an actual column of an array. After careful examination of the source code, what is meant to be a column in this algorithm is contiguous elements of ‘rows’ size as shown below (Figure 7.2). Therefore, future references to column will be understood to mean what was described to be a column in the sentence above.

Within the parameter-calculation operations there are two loops. Each of them traverses one-by-one all the elements of each column. Before each of these loops there is a call to the ‘max_density’ function. This function calculates a parameter referred to in the loops and needs the entire dataset for its execution. The parallelisation effort was initiated on these
two loops. Both of them have similar functionality; hence the same parallelisation strategy is used.

One column of the array is operated in each loop. Eventually, in each loop a column of data is distributed in approximately equal chunks of data as demonstrated in Figure 7.3. The distribution of data is achieved with a combination of a Non-Blocking Send and a Blocking Receive. Prior to the distribution, the master process has to calculate the ‘PMmax’ parameter which is necessary for the operations within the loop. The ‘PMmax’ value is returned by the ‘maxDensity’ function and then broadcasted within the communicator. Other data that are broadcasted consist of the number of rows and the number of columns of the array, so that all the processes allocate the appropriate memory to receive and operate the data.

The output of the operations is an array that contains the elements that fulfil certain conditions as well as the total number of these elements. The operations on each process are easy to be implemented. However, the challenge lies on reassembling the data. Each processor allocates the maximum size that its local array of elements can be. Consequently, only the first elements of the local arrays contain useful information. The master process has to create a unique array that contains the right data in the right order.
In order to implement the correct reassembling procedure, it is important to take under consideration that the local arrays should not be gathered in a contiguous way. In Figure 7.4 we can see how this would result in comparison to the expected outcome. The correct implementation was achieved by applying the following steps:

1. Each worker process stores the elements that fulfil the condition in a local array.
2. At the same time, the workers keep a local counter with the number of the ‘useful’ elements of its local array.
3. The local counters are sent to the master process and are stored in an array.
4. All the workers sent their local arrays. The master receives the local arrays in a single array. Each local array is stored at the position of the master array that equals the size of ‘useful’ elements of the previous process.

At the end of these operations the calculation of the parameters can be done. However, it has to be noted that some functions are executed sequentially by the master process. This includes the functions:

- max_density()
- get_sd()
- get_alpha()

The max_density’s execution seems to be especially important. This function is called twice within the loop hence in order to achieve greater parallelisation this function will have to be also parallelised.

The background adjustment operations take place within another loop. This time all the elements of each column are updated. These calculations need the parameters calculated in the previous steps and are independent for each element. As a result the parallelisation is straightforward. The parameters are stored in an array and broadcasted to all the worker
processes. Thereafter, an approximately equal part of the column is sent to each processor to operate. The results are sent back and stored in the master’s global array.

### 7.2.3 Parallel Background Correction Performance

The Parallel Background Correction implementation seems to have limited scalability. In this section the speedup of this implementation will be calculated and the parallel performance will be evaluated. The behaviour of the new parallel implementation will be under discussion.

The measurements below concern the Background Correction’s execution time. The experiment was executed on NESS and this is a representative result after several executions. At this point, the speedup of the ‘prma_bg_correct’ function will be examined separately from the rest of the RMA function.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Background Correction’s Execution Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46.32</td>
</tr>
<tr>
<td>2</td>
<td>43.25</td>
</tr>
<tr>
<td>4</td>
<td>39.65</td>
</tr>
<tr>
<td>8</td>
<td>40.45</td>
</tr>
<tr>
<td>16</td>
<td>40.71</td>
</tr>
</tbody>
</table>

*Table 7.1 – Parallel Background Correction Performance*

As showed by our measurements the performance gain is poor. In terms of the result’s correctness, the testing application was used and the Expression Set returned is correct when compared to the sequential version’s output.

### 7.2.4 Conclusions

Our data parallel implementation does not achieve satisfactory speedup. It is noticeable that the maximum absolute performance is achieved on four processors. When the ‘prma_bg_correct’ function is executed on more processors, the communication overheads overcome the parallelisation gains.

The number of messages exchanged within the ‘prma_bg_correct’ is considerably large. At the beginning of the calculations the number of rows and columns is broadcasted to the workers. Thereafter, for each column the master has to distribute the data to all the workers (Figure 7.3). Then all the workers send the results of the calculations that take place within the two loops. For each loop these results consist of a counter and an array. Additionally, after the first loop the ‘max_density’ value has to be broadcasted, while after
the second loop the whole parameter array has to be broadcasted. Finally, during the background correction adjustment operations all the workers send the part of the array that they operate back to the master. All these make by default a large total number of messages that becomes even more significant as the number of processors increases.

Another limiting factor for the parallel application’s scalability is the fact that the data are distributed for each column one at a time. In our effort to follow the sequential algorithm’s functionality we operate on each column separately. Consequently, the chunks of data can become relatively small. For the same dataset, when the number of the available processors increases, the chunk size becomes smaller. As an effect of combining a small dataset and a large number of processors, the resulting communication overheads become more important than the performance gained by the parallelisation.

Last but not least, Amdahl’s Law states that the serial parts of a program limit the speedup gained by its parallelisation [18]. As we saw above, there are functions that are called and executed only by the master process.

### 7.3 Data Parallel Approach

As discussed in the beginning of this chapter, we expect that a pure data parallel approach would lead to incorrect results. However, for the sake of research it would be interesting to examine how a data parallel approach could be implemented and how it would perform. It will also be an opportunity to verify our initial assumption that the outcome of this approach will not be correct.

#### 7.3.1 Considering Decomposition Strategies

In data distribution the two factors that should always be taken under consideration are to maintain the load balance among the processors and to minimize the communications among the processors. Two different approaches will be examined and the most appropriate will be implemented.

One method is to divide the dataset so that each processor is assigned with one approximately equal chunk of data. This approach is relatively easy to implement and minimises communication costs. Communication happens twice; when the data is sent to the worker processes and when the processed data is returned back to the master. The number of messages is predefined and equals twice the number of the working processes. However, this may lead to load balance problems since there is not any dynamic load balancing mechanism.

The other is to split the dataset into fixed-sized data chunks. When a process finishes its task, it will be assigned with another chunk of data. In opposition to the previous case load
balance can be guaranteed, especially if we use small-sized chunks of data. However, the smaller the chunk size is, the more messages will have to be sent, increasing the communication overheads.

Assuming that the nature of the algorithm is load balanced, since the same operations will be applied to each of the elements, and taking under consideration that the implementation of the first option will be much more straight-forward we will take this course of action.

### 7.3.2 Applying Parallelisation

As it has already been decided, we will distribute approximately equal chunks of data to all the available processors. Initially all the data are located only on the master process; hence it will have to distribute the data to the workers.

The master process reads the arguments passed to the function while all the other processes have set their corresponding variables to NULL. When the master process gets all the data, it calculates the size of the array object that has to be distributed. Each process will be assigned with an approximately equal number of rows.

The number of the total rows will be divided to the number of the available processors. If the number of rows is not divided by the number of processors the remaining elements will be assigned to the last processor. This way we will split the array into contiguous chunks of data. This is a simple solution, relatively easy to implement and in most cases well balanced.

At this point we would like to note that all the arguments are passed as SEXP pointers. The ‘rma_bg_correct’ function expects a C pointer. Thus, before any data distribution we will transform the SEXP pointer to a numeric pointer. Additionally, before any data distribution all the processors have to allocate some memory to receive the data. Consequently, the master process broadcasts all the information needed which consists of the number of columns, which is the same for every processor, and the number of rows that will be assigned to each processor. Eventually, each processor allocates the space needed for the data to be received and the data can be distributed.

For the distribution of data, a combination of a Non-Blocking Send and a Blocking Receive has been used. With the Non-Blocking send we enable the execution of the program to continue until a specific point of the code without waiting for the message transfer to be completed. This way we can overlap communication and computation and eventually improve the overall performance and avoid deadlocks.

At the time that the message is received, the worker process is ready to call the background correction function on its local dataset. When the calculation finishes each processor sends the results back to the master process. Another combination of a Non-Blocking Send and a Blocking Receive has been used to implement this communication. Finally, the master process allocates the local results of each process to the initial array object according to the
process ID so that the correct order of the data is attained. Eventually, all the data are gathered at the master process which continues the execution of the function while the others exit the execution.

Concluding this section, it is interesting to describe how the data parallel approach call the sequential function in comparison to the parallel version of the background correction paradigm (Listing 7.4). The two models differ fundamentally. While in the first scenario, parallelisation is not applied until the processes enter the ‘prma_bg_correct’ function, in the data parallel model the data are distributed before the processes call the ‘rma_bg_correct’ function.

<table>
<thead>
<tr>
<th>Parallel Background Correction</th>
<th>Data Parallel Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (worldRank == 0)</td>
<td>//Distribute PM array to Local PMx arrays</td>
</tr>
<tr>
<td>{</td>
<td>...</td>
</tr>
<tr>
<td>prma_bg_correct(0,PM, rows, cols);</td>
<td>rma_bg_correct(PMx, rowspp, cols);</td>
</tr>
<tr>
<td>}</td>
<td>//Reassemble the PM array</td>
</tr>
<tr>
<td>else</td>
<td>...</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>prma_bg_correct(0);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Listing 7.4 – Background Correction Call

7.3.3 Data Parallel Performance
In order to evaluate our results better we will measure both the time taken for the background correction to be executed and the total execution time of the prma function.

We calculated the speedup of the parallel code so that we have some metrics of the prma’s performance, where speedup is defined as the result of the quotient of the sequential execution time to the parallel execution time:

\[
Sp = \frac{T_1}{Tp}
\]

In the table below (Table 7.2) the execution time of the prma on a different number of processors on the NESS system is demonstrated. In each case the execution time is determined by the execution on the slowest processor adding any communication
overheads due to message passing. However, the communication costs are not expected to create big overhead since there is not a large number of messages exchanged. Message passing includes only the distribution of the data and then the gathering to the master process. It is noticeable that the communication costs are slightly smaller as the number of the processors increases and eventually the message size decreases.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Slowest Processor Time (secs)</th>
<th>Total Execution Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N/A</td>
<td>45.19</td>
</tr>
<tr>
<td>2</td>
<td>30.29</td>
<td>31.17</td>
</tr>
<tr>
<td>4</td>
<td>23.36</td>
<td>24.44</td>
</tr>
<tr>
<td>8</td>
<td>19.78</td>
<td>20.70</td>
</tr>
<tr>
<td>16</td>
<td>18.39</td>
<td>19.17</td>
</tr>
</tbody>
</table>

*Table 7.2 – Data Parallel Performance*

In the following figure the background correction’s speedup is shown in comparison to a linear speedup. We can see that even though there is improvement in the absolute execution time the speedup is not satisfactory. Especially as the number of the available processors increases, the program does not scale efficiently. Although each processor is assigned with smaller datasets that does not mean that the execution time is proportionally less. This restriction is determined by the serial code that we execute; in our case the ‘rma_bg_correct’ function. In data parallel solutions this is a restriction that has to be taken under consideration.
However, we should not only focus on a specific part of the code. We have to investigate how our parallel version of the background correction function affects the execution of the program in its entirety.

Figure 7.5 – Background Correction (Data Parallel) Speedup

Figure 7.6 – PRMA Speedup
7.3.4 Conclusions
The data parallel implementation is proven to have limited scalability. Even when the ‘rma_bg_correct’ function’s scalability was examined separately the results were not very impressive. In Figure 7.5 we saw that the speedup is limited. However, the same speedup seems important in comparison to the prma’s speedup (Figure 7.6). As we mentioned in Chapter 3, where we analysed the RMA function, the background correction part consists approximately of the 50% of the total execution. According to Amdahl’s law, the performance to be gained by the parallelisation of a program is restrained by its sequential part. Unfortunately, it is easily perceived that these restrictions apply in our case.

Eventually, in order to have better results parallelisation has to be applied in other parts of the code as well. This part will be determined by the profiling results that we already have as well the parallelisation potentials for that.
Chapter 8 - Normalisation

According to our profiling results the most important function in terms of execution time, after the background correction, is the normalisation function. In this chapter we will focus on the potential parallelisation strategies of this function and the implementation of a new parallel function.

8.1 Considering Normalisation Parallelisation

In Chapter 3, the functionality of the normalisation procedure was described as a sequence of calculations that involve operations that need access to all the elements of the object. Eventually, similar to the background correction function a data parallel approach cannot be applied without affecting the final result. In case of a data parallel implementation we could have a normalised sample but this would be different than the sequential outcome. Consequently, this could not be an acceptable solution. In order to parallelise this part of the prma function we have to parallelise the normalisation function itself instead of distributing the data and executing the already existing sequential function.

The execution flow of the program as it is after the parallelisation of the background correction is not suitable for parallelising the normalisation function as well. As we can see in the ‘Listing 8.1’ the other processes return while only the master process continues its execution. More specifically, the master process calls the rma_c_call function. Within this function, the data are initially prepared for the call of the normalisation function. Eventually, the qnorm_c function is called. Finally, the expression set is calculated, which is not considered to be an overhead, and the result is returned to the rma_c function.

However, in order to have all the processes involved in the normalisation function the code has to be altered. At first, the rma_c_call function has to be called by all the available processes and since the data are only available on the master process the argument list has to be dynamic. While the master process calls the rma_c_call function with an argument list, all the others call the same function without any arguments.
Respectively, within the rma_c_call function all the processes have to call the normalisation function. However, before the call of the normalisation function we have to take under consideration the Normalisation Flag’s value. This argument determines whether the normalisation function will be executed or not. Eventually, the value of the flag has to be broadcasted to all the processes. Only if the flag is set to true, the processes will enter the normalisation function. The master will pass the argument list while the workers will only call the function.
Applying Normalisation Parallelisation

The ‘qnorm_c’ function includes a POSIX thread implementation similar to that of the background correction function. The need for faster execution of this function has been encountered in the past. However, for the same reasons as in the background correction case which includes incompatibility with SPRINT and availability only on shared memory machines, this implementation has not proved itself useful in terms of our project. Any POSIX reference has been removed and the development of the MPI parallel version started from a pure sequential code.

The serial source code consists of two function calls which implement the four steps of the Normalization process (Section 3.2). The ‘normalize_determine_target’ function implements the two first steps, which include sorting the array columns and calculating the average of each row. Our parallelisation efforts will focus on that function. The fact that the calculations within this function refer to different dimensions of the array in each step makes its parallelisation more challenging. The second function is the ‘normalize_distribute_target’ and implements the rest of the operations which consist of replacing the elements of each row with the average calculated and returning the elements into their initial positions. It is worthwhile to mention that the sequential execution of the normalisation calculations took 20.8 seconds of which 7.55 were on ‘normalize_determine_target’ and 13.26 on ‘normalize_distribute_target’.

The data parsing is done in a similar way to the background correction function. The convention made in the background correction function about the meaning of the term ‘column’ still applies in this case. A column is considered to be a set of ‘rows’ number of elements. However a different decomposition strategy is applied now. The columns are distributed in a cyclical way among the worker processors. A visual representation is given in the following figure. This is an example of three worker processes (P1, P2, P3) operating on a 4x8 array.

![Cyclical Data Distribution](image)
For each column the ‘qsort’ function is applied. The mean value for each row is calculated and stored in a row sized array. The worker processes send their local arrays to the master process which calculates a global row mean array.

### 8.3 Parallelisation Performance

As mentioned before, in this case a new decomposition strategy was followed, unlike within the background correction scenario. Because of the poor performance that we had before, it would be interesting if another approach could lead to more efficient outcomes.

In order to have a clear result of how our last parallelisation effort performs, we timed the ‘normalize_determine_target’ individually. As shown in the table below the cyclical distribution schema seems to add some overheads for small number of processors. It is noticeable that the execution of the parallel function on one processor takes more time than the original sequential function. However, as the available worker processors increase, the speedup becomes more satisfactory. Once more, the size of the dataset seems to be a limiting factor for the parallel applications scalability. After a specific point the number of processes rise and the scalability of the application is limited. Our measurements took place on NESS and the measurement presented is a representative sample among several executions.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Execution Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.16</td>
</tr>
<tr>
<td>2</td>
<td>8.46</td>
</tr>
<tr>
<td>4</td>
<td>4.47</td>
</tr>
<tr>
<td>8</td>
<td>2.58</td>
</tr>
<tr>
<td>16</td>
<td>2.43</td>
</tr>
</tbody>
</table>

*Table 8.1 – Determine Target Performance*

Even though there are restrictions on the scalability of the parallelisation of this part of the code, it seems that the speedup in this case is satisfactory. However, only part of the normalisation function is parallelised. As explained before, according to Amdahl’s Law, the sequential part of the function restricts the speedup gained. In Figure 8.3 the ‘determine target’s’ speedup is compared to the speedup of the whole normalisation function. This case is precisely illustrated in Figure 8.3 below. The speedup of the entire normalisation function is not as significant as its parallelised part but we can see how it is positively improved by the individual part.
8.4 Conclusions

In this case we achieved the best results in comparison to our other parallelisation approaches. Distributing larger chunks of data on each processor was beneficial for the performance of the parallel application. Although, it seems that for small size of the dataset the scalability is limited, it is our belief that it is not of great importance. Biostatistical research deals with datasets larger than our test dataset. Hence, for reasonable number of processors the speedup gain could be effective. On the other hand, the main limiting factor is the sequential part of the normalisation function. In order to further improve the overall normalisation speedup the rest of the normalisation function has to be parallelised as well.
Chapter 9 Overall Conclusions

The final chapter of this report aspires to summarise all the partial conclusions that were made after each chapter, in order to give to the reader a general perspective of the entire project. Additionally, we will try to evaluate whether or not the original aims of the project were satisfied and future work suggestions will be made. Finally, a reference to the risks encountered will be made and the initial risk estimation that had been done will be evaluated.

9.1 Results Evaluation

The original aim of this project was to investigate the parallelisation opportunities of the RMA function as part of the SPRINT project, as we could not know in advance whether parallelisation would be achievable. More specifically the project proposal of this thesis targeted on [19]:

- research in parallelising the rma() function
- come to several prototype solutions
- investigate the possibility of implementing one or more of them

The possibility of the project not resulting in a working parallelised RMA function was taken into consideration. The alternatives were to parallelise only some fractions of the code or in the worst case scenario, only provide suggestions of potential parallelisation.

During the project, we had the opportunity to examine the two main approaches of parallelisation that are used within SPRINT. The main overhead in RMA’s execution was observed firstly at the background correction and then at the normalisation operations. Each of these functions was tackled separately. For the sake of our research we tried to apply different decomposition models and evaluate their performance.

Since it seemed to be more important in terms of execution time, the initial point of our parallelisation efforts was the background correction. Even though the data parallel background correction approach was expected to return an altered outcome in comparison to the sequential function, we decided to proceed with its implementation in the interest of furthering the research. There was some gain in the performance of this implementation, but it was not very impressive and its scalability seemed to be limited. In conjunction with the incorrect results that the data parallel approach returned, the need for a new parallel function arose. Applying parallelisation in this case was more challenging. This approach
required more profound knowledge of the sequential source code and specifically in the dataset’s flow during the execution. The decomposition strategy that seemed more suitable at the time of the implementation was to distribute equal subsets of each column and operate on them. This procedure would be executed for each column of the dataset. Unfortunately, this implementation performed worse than the data parallel although in this case the results were returned correctly. The communication overheads and especially the fact that the chunk of data operated at a specific time did not require very significant computational resources, limited the performance gain and the scalability of the parallel application. Additionally, an important limiting factor as presented in the relevant section is the existence of important sequential parts.

Having the experience of the background correction, we proceeded to the normalisation where we focused on one of the two main functions called within this procedure. Since the performance in the previous attempts was poor, it was decided to implement a parallel approach with another decomposition model. This time each processor was assigned with one column in a circular way to ensure the load balance among the workers. Although for small number of working processors there was not any performance gain, as the number of processors increased the parallel application was more efficient. Eventually, as the number of processors increases too much, the scalability meets its limits. However, in comparison to all the other approaches that we experimented on, this was the solution that returned the best results.

![Figure 9.1 – Speedup Comparison](image-url)
Evaluating our outcome, despite the fact that the overall performance improvement achieved is not impressive, the fundamental objective of the project is accomplished. A detailed investigation of the parallelisation opportunities was made. This consists of examining the parallelisation approaches that are supported by SPRINT as well as examining different decomposition strategies in each of our cases.

Finally, after having applied different parallelisation approaches, we concluded that the cyclical decomposition strategy that was used in the normalisation case-study performs the most satisfactorily.

9.2 Risks Encountered
At the beginning of the project a list of the imminent risks had been made. This included the lack of prior experience on the SPRINT project, the absence of any biostatistical background and the lack of sufficient R and SPRINT documentation which actually were proven to be the most important obstacles hindering this project.

Initially, there was much time spent investigating the R framework. Although this lead to the creation of a stable background on R, it was not proven to be that useful. Most of the implementation of the parallel version was made in C and basic R knowledge would be sufficient. Here ‘basic knowledge’ mostly refers to being familiar with the extension mechanism of R and a basic understanding of the R data objects.

Moreover, it became apparent that more time than was actually necessary was dedicated to building a biostatistical background. It is our belief that instead of trying to understand what the RMA function does, it would have been more beneficial to examine how it does it and how the dataset is affected.

In addition, the time spent to set a sequential version of RMA within SPRINT was more than the initial estimation made at the outset of the project. Isolating the RMA function from the affy package was complicated due to the dependancies to the preProcess package.

Last but not least, the limited developer’s documentation of the SPRINT project was covered by the brief but useful SPRINT developer’s guide provided by Michal Piotrowski.

9.3 Future Work Suggestions
Unfortunately, our prma implementation does not have a satisfactory parallel performance. However, taking advantage of the findings of our research can lead to an efficient future implementation. In our parallelisation attempts the main overheads derive firstly from the existence of significant sequential parts. Furthermore, in the background correction case the distribution model was proven to be inefficient.
According to our findings, future implementations of the background correction function should prioritise a focus on applying the cyclical decomposition that was used in the normalisation case-study. Apart from that implementing parallel versions of the sequential functions that are still used within background correction will lead to further improvement of the parallel performance.

As far as the normalisation function is concerned, the 'normalize_distribute_target' has also to be parallelised. At the moment its sequential execution limits the speedup of the parallel application.

Finally, apart from making the RMA function be executed faster, it is important to enable its execution in cases that due to high memory demands this is not possible. When considering the parallelisation approaches that we implemented, there was also a thought of applying the parallel input/output supported by MPI2. Due to the design of the algorithm the dataset is distributed and gathered during the execution of the application too many times. The overheads that would be caused due to writing and reading from a file were considered to be a limiting factor for the parallel performance and this idea was abandoned. However, it would be interesting to implement this approach in practice and evaluate the gains that occur as a result.
Appendix A
Profiling – Benchmarking

R Profiler Output for RMA
Each sample represents 0.02 seconds.

Total run time: 93.64 seconds.
Total seconds: time spent in function and callees.
Self seconds: time spent in function alone.

<table>
<thead>
<tr>
<th>%</th>
<th>self</th>
<th>% total</th>
<th>self seconds</th>
<th>total seconds</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>85.4</td>
<td>80.00</td>
<td>85.4</td>
<td>80.00</td>
<td>&quot;Call&quot;</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>6.24</td>
<td>13.6</td>
<td>12.74</td>
<td>&quot;local&quot;</td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>1.56</td>
<td>1.7</td>
<td>1.56</td>
<td>&quot;ls&quot;</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>1.44</td>
<td>1.5</td>
<td>1.44</td>
<td>&quot;load&quot;</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>1.32</td>
<td>1.4</td>
<td>1.32</td>
<td>&quot;as.vector&quot;</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>0.98</td>
<td>1.1</td>
<td>0.98</td>
<td>&quot;FUN&quot;</td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td>0.52</td>
<td>0.6</td>
<td>0.52</td>
<td>&quot;order&quot;</td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td>0.36</td>
<td>1.4</td>
<td>1.34</td>
<td>&quot;lapply&quot;</td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>0.28</td>
<td>0.3</td>
<td>0.28</td>
<td>&quot;mget&quot;</td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>0.22</td>
<td>1.5</td>
<td>1.42</td>
<td>&quot;unlist&quot;</td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>0.18</td>
<td>0.2</td>
<td>0.18</td>
<td>&quot;sort.list&quot;</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>0.08</td>
<td>0.1</td>
<td>0.08</td>
<td>&quot;as.list.environment&quot;</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>0.06</td>
<td>0.1</td>
<td>0.12</td>
<td>&quot;match&quot;</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>0.06</td>
<td>0.1</td>
<td>0.06</td>
<td>&quot;is.na&quot;</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>0.06</td>
<td>0.1</td>
<td>0.06</td>
<td>&quot;list&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.04</td>
<td>0.4</td>
<td>0.34</td>
<td>&quot;split.default&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.04</td>
<td>0.0</td>
<td>0.04</td>
<td>&quot;unique.default&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.02</td>
<td>0.0</td>
<td>0.02</td>
<td>&quot;NROW&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.02</td>
<td>0.0</td>
<td>0.02</td>
<td>&quot;colnames&lt;-&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.02</td>
<td>0.0</td>
<td>0.02</td>
<td>&quot;eapply&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.02</td>
<td>0.0</td>
<td>0.02</td>
<td>&quot;is&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.02</td>
<td>0.0</td>
<td>0.02</td>
<td>&quot;lazyLoadDBfetch&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.02</td>
<td>0.0</td>
<td>0.02</td>
<td>&quot;levels&lt;-&quot;</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.02</td>
<td>0.0</td>
<td>0.02</td>
<td>&quot;selectMethod&quot;</td>
<td></td>
</tr>
</tbody>
</table>
### Background Correction – Data Parallel

#### On 1 Processor

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time on processor 0 including overheads</td>
<td>45.195962</td>
</tr>
<tr>
<td>Execution Time on processor 0</td>
<td>44.423867</td>
</tr>
</tbody>
</table>

#### On 2 Processors

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time on processor 0 including overheads</td>
<td>31.178709</td>
</tr>
<tr>
<td>Execution Time on processor 0</td>
<td>29.626552</td>
</tr>
<tr>
<td>Execution Time on processor 1</td>
<td>30.296349</td>
</tr>
</tbody>
</table>

#### On 4 Processors

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time on processor 0 including overheads</td>
<td>24.443810</td>
</tr>
<tr>
<td>Execution Time on processor 0</td>
<td>23.360062</td>
</tr>
<tr>
<td>Execution Time on processor 1</td>
<td>23.090428</td>
</tr>
<tr>
<td>Execution Time on processor 2</td>
<td>22.841440</td>
</tr>
<tr>
<td>Execution Time on processor 3</td>
<td>22.920397</td>
</tr>
</tbody>
</table>

#### On 8 Processors

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time on processor 0 including overheads</td>
<td>20.709873</td>
</tr>
<tr>
<td>Execution Time on processor 0</td>
<td>19.093463</td>
</tr>
<tr>
<td>Execution Time on processor 1</td>
<td>19.597873</td>
</tr>
<tr>
<td>Execution Time on processor 2</td>
<td>19.763724</td>
</tr>
<tr>
<td>Execution Time on processor 3</td>
<td>19.604785</td>
</tr>
<tr>
<td>Execution Time on processor 4</td>
<td>19.788595</td>
</tr>
<tr>
<td>Execution Time on processor 5</td>
<td>19.485071</td>
</tr>
<tr>
<td>Execution Time on processor 6</td>
<td>19.479615</td>
</tr>
<tr>
<td>Execution Time on processor 7</td>
<td>19.362076</td>
</tr>
</tbody>
</table>

#### On 16 Processors

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time on processor 0 including overheads</td>
<td>19.172271</td>
</tr>
<tr>
<td>Execution Time on processor 0</td>
<td>18.172439</td>
</tr>
<tr>
<td>Execution Time on processor 1</td>
<td>17.982537</td>
</tr>
<tr>
<td>Execution Time on processor 2</td>
<td>18.040294</td>
</tr>
<tr>
<td>Execution Time on processor 3</td>
<td>18.036163</td>
</tr>
<tr>
<td>Execution Time on processor 4</td>
<td>17.980793</td>
</tr>
<tr>
<td>Execution Time on processor</td>
<td>Time</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>5</td>
<td>17.970462</td>
</tr>
<tr>
<td>6</td>
<td>18.014506</td>
</tr>
<tr>
<td>7</td>
<td>18.219212</td>
</tr>
<tr>
<td>8</td>
<td>18.039196</td>
</tr>
<tr>
<td>9</td>
<td>18.046581</td>
</tr>
<tr>
<td>10</td>
<td>18.348789</td>
</tr>
<tr>
<td>11</td>
<td>17.960847</td>
</tr>
<tr>
<td>12</td>
<td>18.224757</td>
</tr>
<tr>
<td>13</td>
<td>18.084017</td>
</tr>
<tr>
<td>14</td>
<td>18.398206</td>
</tr>
<tr>
<td>15</td>
<td>18.023092</td>
</tr>
</tbody>
</table>
Appendix B
Code Fractions

The Namespace File source code

```c
#ifndef _COMMANDS_H
#define _COMMANDS_H

/**
 * Lists all the functions available, ensure that TERMINATE is first and
 * LAST is last. If you add a command code you must add a command function
 * in sprint/functions.c
 **/

typedef int (*commandFunction)(int n,...);

#endif
```

Function.h source code

```c
#ifndef  _COMMANDS_H
#define  _COMMANDS_H

/**
 * Lists all the functions available, ensure that TERMINATE is first and
 **
 * LAST is last. If you add a command code you must add a command function
 * in sprint/functions.c
 **/

typedef int (*commandFunction)(int n,...);

#endif
```
The Data Parallel Implementation

```c
SEXP rma_c(int n, ...){
    va_list ap; //will point to each unnamed argument in turn
    SEXP PMmat = NULL;
    SEXP ProbeNamesVec = NULL;
    SEXP N_probes = NULL;
    SEXP norm_flag = NULL;
    SEXP bg_type = NULL;
    SEXP verbose = NULL;
    SEXP result = NULL;
    double *PM = NULL;
    int worldSize, worldRank;
    SEXP dim1;
    int rows, cols, i, rowspp;
    double *PMx;

    // Get size and rank from communicator
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
    MPI_Request req[worldSize];
    MPI_Status status;

    if (worldRank == 0)
    {
        // Get input variables
        va_start(ap,n);
        PMmat = va_arg(ap,SEXP);
        ProbeNamesVec = va_arg(ap,SEXP);
        N_probes = va_arg(ap,SEXP);
        norm_flag = va_arg(ap,SEXP);
        bg_type = va_arg(ap,SEXP);
        verbose = va_arg(ap,SEXP);
        va_end(ap);

        PROTECT(dim1 = getAttrib(PMmat,R_DimSymbol));
        rows = INTEGER(dim1)[0];
        cols = INTEGER(dim1)[1];
        PM = NUMERIC_POINTER(PMmat);
        rowspp = rows/worldSize;
        UNPROTECT(1);
    }

    //BROADCAST DATA NEEDED FOR ALL PROCS
    MPI_Bcast(&rowspp, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //MASTER process sends chunks of data to WORKERS
    if (worldRank==0)
```
{  
    for (i=0; i<worldSize; i++)  
    {  
        MPI_Issend(&PM[i*rowsspp*cols], rowsspp*cols, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &req[i]);  
    }  
}

//Allocate space on each process and receive data
PMx = (double *)malloc(sizeof(double) * rowsspp*cols);
MPI_Recv(&PMx[0], rowsspp*cols, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

if (worldRank==0)  
{  
    for (i=0; i<worldSize; i++)  
        MPI_Wait(&req[i], &status);  
}

//Master Prints Message
if (worldRank==0)  
{  
    if (INTEGER(verbose)[0])  
        Rprintf("Background correcting\n");  
}

//Every PROCESS executes backGround Correction
rma_bg_correct(PMx, rowsspp, cols);

//WORKERS send data to MASTER
MPI_Issend(&PMx[0], rowsspp*cols, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req[worldRank]);
if (worldRank==0)  
{  
    for (i=0; i<worldSize; i++)  
    {  
        MPI_Recv(&PM[i*rowsspp*cols], rowsspp*cols, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);  
    }  
    MPI_Wait(&req[worldRank], &status);  
}

free(PMx);

//MASTER PROCESS executes normalisation
if (worldRank==0)  
{  
    result = rma_c_call(0, PMmat, ProbeNamesVec, N_probes,norm_flag,verbose);  
}
else
{
    rma_c_call(0);
}

return result;
Bibliography


[3] R Development Core Team: Writing R Extensions

[4] .Call function help page


[9] RMA function R help page

[10] Ben Bolstad: Microarray Data Processing for Affymetrix arrays


[16] Lawrence Mitchell: A parallel Random Forest implementation for R

[17] Savvas Petrou: SPRINTing with HECToR


[19] Spilios Alexopoulos: Project Preparation Coursework