Improving the Computing Efficiency of the Regional Genomic Relationship Mapping Approach

Josaphat Cuevas Ortuno

August 24, 2012
# Contents

1 Introduction 1

2 Background theory 3

2.1 Regional Genomic Relationship Mapping approach 3
  2.1.1 Genomic and regional heritability 4
  2.1.2 Regions or windows of the genome 4

2.2 General Linear Mixed Model 5
  2.2.1 Mixed Model fitted by the RGRM approach 6

2.3 Software Implementation 7
  2.3.1 Suite of Programs 7
  2.3.2 Computing Challenges 8
  2.3.3 Runtime Environment 9

2.4 Optimisation of ASReml 10
  2.4.1 Incremental Parallelisation 10
  2.4.2 Serial and Parallel Versions 10
  2.4.3 Parallelisation Within a Node 11

2.5 OpenMP 11
  2.5.1 Programming Interface 11
  2.5.2 Loop-level Parallelism 13
  2.5.3 Loop Scheduling 13

3 ASReml 15

3.1 Input Data 15

3.2 Mapping of Algorithm to Source Code 15
  3.2.1 Code Inspection 16
  3.2.2 Fortran Procedures of AI REML iteration 18

3.3 Sparse Matrices Storage 19
  3.3.1 Sparse Matrix Storage Format 19

3.4 Operations on Elements of the Sparse Satrix 20
  3.4.1 READ an entire row and its Diagonal Element 21
  3.4.2 READ the value of a row’s column 21
  3.4.3 UPDATE the value of a row’s column 21
  3.4.4 INSERT a column’s row 23

3.5 Profiling of Serial Version 23
3.5.1 Experimental Results ........................................ 23
3.5.2 Discussion of experimental results .......................... 28

4 Parallelisation of ASReml using OpenMP ...................... 29
  4.1 Revision of Memory Data Structures .......................... 29
  4.2 Improve of code readability and maintainability ............. 29
  4.3 Description of Changes ........................................ 30
List of Tables

2.1 List of programs involved in the analysis of each window of the genome. 7
2.2 Computing nodes offered by the current installation of Eddie. 9
2.3 CPUs installed on the nodes of Eddie. 10
2.4 Fortran routine prototypes exposed by ’omp.h’. 12
2.5 Some environment variables meaningful to the OpenMP runtime. 13

3.1 Input data files used to run our experiments with ASReml. 16
3.2 Steps of ASReml iteration controlled by G5VITR subroutine. 18
3.3 Calls made by G5VXXA and G5VWVA subroutines to G5VLLI and G5VXIJ. 24

4.1 Routines defined in g5vxxa_sparsefinal.f source code file. 30
4.2 Routines defined in g5vxxa_densefinal.f source code file. 30
4.3 Routines defined in g5vwva_sparsefinal.f source code file. 31
4.4 Routines defined in g5vwva_densefinal.f source code file. 32
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Overlapping of regions of the genome: windows of 100 adjacent SNPs shifted every 50 SNPs in a genome-wide data array of 275,000 SNPs.</td>
</tr>
<tr>
<td>2.2</td>
<td>Interaction of suite of programs to analyse a region of the genome</td>
</tr>
<tr>
<td>3.1</td>
<td>Linked list implemented as a two dimensional array (IX) to store the sparse matrix shape</td>
</tr>
<tr>
<td>3.2</td>
<td>Representation of the sparse storage index in ASReml</td>
</tr>
<tr>
<td>3.3</td>
<td>Sparse matrix formed by ASReml with 22 equations (2 dense and 20 sparse)</td>
</tr>
<tr>
<td>3.4</td>
<td>Reading a row and its diagonal element by G5VXIL routine</td>
</tr>
<tr>
<td>3.5</td>
<td>Reading the value of a row’s column by GVXIJ routine</td>
</tr>
<tr>
<td>3.6</td>
<td>Updating the value of a row’s column by G5VLLI routine</td>
</tr>
<tr>
<td>3.7</td>
<td>Inserting a row’s column by G5VLLI routine</td>
</tr>
<tr>
<td>3.8</td>
<td>Fraction of total execution per routine with -O0 and -O3 compiler optimisation</td>
</tr>
<tr>
<td>3.9</td>
<td>Fraction of total execution per routine with -O0 and -O3 compiler optimisation</td>
</tr>
<tr>
<td>3.10</td>
<td>Consolidation of fractions of total execution times taken by ASReml subroutines</td>
</tr>
<tr>
<td>3.11</td>
<td>Timing of ASReml iterations with/without reordering of the matrix</td>
</tr>
<tr>
<td>3.12</td>
<td>Timing of ASReml iterations with/without reordering of the matrix</td>
</tr>
<tr>
<td>3.13</td>
<td>Average times of iterations 2 to 6 taken by iteration steps with/without matrix reordering</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The Medical Research Council (MRC) Human Genetics Unit in the United Kingdom is a leader research centre in human genetics. It was established in 1967 and it is located in the Western General Hospital in Edinburgh. Researches developed by the Unit focus on the understanding of genetic factors implicated in human diseases and normal and abnormal development and physiology. Some of the areas of work in the centre are the developmental genetics, common disease genetics, chromosome biology and models for human genetic diseases.

A group of geneticists in the Unit have developed a new approach that seems to overcome the limitations of traditional single-SNP and linkage analyses. The new approach is called the regional genomic relationship mapping (RGRM). Experimental results of analyses of real and simulate data using the RGRM approach have proved its effectiveness, but it is computationally demanding. It has has limited the power of the geneticists to undertake more analysis.

Faced by these computing limitations, the team in the MRC Human Genetics Unit have looked for the collaboration with the Edinburgh Parallel Computing Center (EPCC) of the University of Edinburgh in order to explore possibilities in improving the computing efficiency of the RGRM approach. It led to set up a collaboration project between the MRC in Human Genetics Unit and the EPCC that could be undertaken by one of the MSc in High Performance Computing students as the dissertation work. This dissertation has been led by Eilidh Troup and Iain Bethune as project supervisor and co-supervisor, respectively, both members of the EPCC staff. From the MRC Humans Genetic Unit, Pau Navarro and Ricardo Pong-Wong have been the researchers involved in the project and main contacts of the institute.

The current implementation of the RGRM approach is a set of programs written in R, Fortran and shell scripting and they are executed in a defined order by a set of scripts. Each of this program implement routines that map to the steps needed by the RGRM algorithm. The MRC Human Genetics was able to identify the steps and programs that were taking most of the computing time during each analysis. The analysis of data by using restricted maximum likelihood (REML), one step of the approach, resulted to be
the most computing demanding part of the process. The REML analysis is implemented by the ASReml software, a commercial software written in Fortran 77 that is developed and maintain by VSN International, a company dedicated to develop software for bio-science. The program was originally developed by Arthur Gilmour. The version of the program used by the researchers is an unlicensed version which has allowed to access the source code and the implementation details.

The scope of this dissertation has been the analysis of the unlicensed implementation of the ASReml program and the exploration of opportunities of increasing its computing efficiency by using parallel computing techniques and efficient programming principles in order to speed up the performance of the program.

The result of the work of this dissertation has been the delivery of a new version of the ASReml program that implements OpenMP to speed up the computing efficiency of the critical parts of the program. A description of the reasoning for choosing OpenMP as the option to parallelise the existing code and a detailed description of the changes made to the code are explained on this report.

The new parallel version of the program has proved to increase the computing efficiency of the RGRM approach by reducing the overall execution time. Several benchmarks applied to the serial version and the parallel version aim to compare the success of the new implementation in terms of execution time, parallel speed up and parallel efficiency.
Chapter 2

Background theory

2.1 Regional Genomic Relationship Mapping approach

The identification of new loci (plural of locus, position of a gene on a chromosome) that contribute to complex human trait variations such as height or weight and to the predisposition to diseases such as cancer, diabetes or schizophrenia in a large group of individuals has been possible due to recent advances in genetics. These studies are based on the analysis of associations between particular variants at genetic markers and trait variation in large groups of population genotyped. The genetic markers used for this purpose are single nucleotide polymorphisms (SNPs), which are the most common type of genetic variation among humans. Nucleotides are the building blocks of the Deoxyribonucleic Acid (DNA) and they are referred with the letters A (adenosine), C (cytosine), G (guanosine) and T (thymidine). A SNP is found when a single nucleotide is different on a chromosome of two different individuals.

Genome-wide association studies (GWAS) have been widely applied for this purpose and they have proved their success in understanding the genetic factors of many complex traits. For some diseases and traits, such as diabetes type 1 and 2, prostate cancer, breast cancer, height and fat mass the database of new loci discovered has rapidly increased in recent years, but for most traits and diseases only a low proportion of the genetic variation have been identified by standard GWAS. GWAS are single-SNP analyses in the whole genome and they search for SNPs that are present more frequently in a group of individuals with a particular trait or disease (the case group) and a different group of individuals without it (the control or healthy group). The results of this studies are the identification of loci with the highest risk factor to contribute to the presence of the trait or disease which is subject of study.

Based on the fact that GWAS have only found a low number of loci associated with most complex traits and diseases, a research in the MRC Human Genetics Unit, suggests that it reflects the limited power of single SNP analysis in order to detect rare causative alleles (different forms of a genetic locus) or those of small effects that individually contribute to a small variation but collectively may contribute a substantial proportion
of heritability. Researchers in the Unit have developed a new methodology or approach named the regional genomic relationship mapping (RGRM) approach and it promises to overcome the limitation of traditional GWAS. The RGRM approach combines the ability of linkage analysis with the ability of SNP based associations to determine the variance in the population of study. Studies based on real and simulation data have verified the effectiveness of the RGRM approach compared to GWAS but it has also outlined that this approach is computationally demanding.

2.1.1 Genomic and regional heritability

The RGRM approach uses genome-wide SNP data in order to estimate the genetic relationship between all pairs of individuals in the population of study. Two different genetic relationships are estimated: The genomic heritability, which represent the trait variance contributed by the genetic relationship of the whole genome and the regional heritability that represents the contribution to trait variants of each region of the genome.

In order to estimate these heritabilities, a genome-wide SNP array of data is used to obtain the genome-wide relationship matrix and the regional relationship matrices. The regional relationship matrices are identity by descent (IBD) matrices and there is one for each region of the genome and they represent sequential regions of SNPs across the genome. Finally, using the matrices obtained, the genomic and regional heritability are estimated fitting an statistical linear mixed model which estimates the variance parameters using a restricted-maximum likelihood (REML) analysis.

2.1.2 Regions or windows of the genome

The RGRM approach is able to estimate the regional heritability by partitioning the whole genome into small regions or windows containing adjacent SNPs. A study using the RGRM approach analysed a population of 3000 individuals from an European population looking for genetic variations that contributed to serum uric concentration and height traits using a genome-wide SNP data array of 275,000 SNPs. It used different windows sizes of 10, 20 and 100 SNPs to estimate regional heritabilities and the results demonstrated that a smaller size of windows improved the mapping resolution in the genetic regions of the genome. The study also explored the overlapping of windows (size of 100 SNPs) shifting the windows with every 50 SNPs. It translated to 5501 windows and to the same number of analyses. Figure 2.1 shows how the windows of 100 SNPs were overlapped and shifted every 50 SNPs for this study.
Figure 2.1: Overlapping of regions of the genome: windows of 100 adjacent SNPs shifted every 50 SNPs in a genome-wide data array of 275,000 SNPs.

### 2.2 General Linear Mixed Model

The regional heritability associated with each region of the genome and the whole genomic heritability are estimated by fitting an statistical general linear mixed model. Mixed models have been applied in genetic analysis and the estimation of the variance parameters in linear mixed models by REML has been widely adopted as an efficient solution. A linear mixed model or lineal mixed effects model, contains both, fixed effects and random effects. It is described by the equation:

\[
y = X\beta + Zu + \epsilon
\]  

(2.1)

where

- \(y\) is the vector of observations of size \(n\) \((y_1...y_n)\)
- \(X\) is a \((n \times p)\) design matrix for the vector of fixed effects \(\beta\) of size \(p\)
- \(Z\) is a \((n \times q)\) design matrix for the vector of random effects \(u\) of size \(q\)
- \(\epsilon\) is the vector of residual errors of size \(n\) \((\epsilon_1...\epsilon_n)\)

The mean and variance of the vectors for fixed random effects and residual errors are defined by:

\[
\begin{bmatrix} u \\ e \end{bmatrix} \sim N \left( \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \sigma^2 \begin{bmatrix} G(\gamma) & 0 \\ 0 & R(\phi) \end{bmatrix} \right)
\]  

(2.2)
it can also be expressed as

\[ u \sim (0, \sigma^2 G), \epsilon \sim (0, \sigma^2 R) \]  

(2.3)

where \( G \) and \( R \) matrices are functions of the parameters \( \gamma \) and \( \phi \). Given \( G \) and \( R \) matrices, the solutions for \( \beta \) and \( u \) are determined by solving the mixed model equations (MME):

\[
\begin{bmatrix}
X'R^{-1}X & X'R^{-1}Z \\
Z'R^{-1}X & Z'R^{-1}Z + G^{-1}
\end{bmatrix}
\begin{bmatrix}
\tilde{\beta} \\
\tilde{u}
\end{bmatrix}
= 
\begin{bmatrix}
X'R^{-1}y \\
Z'R^{-1}y
\end{bmatrix}
\]

(2.4)

For solving these MME, it is necessary to obtain the values for \( \gamma \) and \( \phi \). These values are replaced by their REML estimates.

### 2.2.1 Mixed Model fitted by the RGRM approach

The whole genomic and regional heritabilities are estimated by a variance component analysis. The linear mixed model fitted is expressed by the equation:

\[ y = X\beta + Zu + Zv + \epsilon \]  

(2.5)

where

- \( y \) are the phenotypic values (serum acid concentration and height)
- \( X \) is a design matrix for fixed effects \( B \)
- \( B \) is the vector of the fixed effects of sex, population, village and age of the population genotyped
- \( Z \) is a design matrix for the random effects \( u \) and \( v \)
- \( u \) is the vector of random effects of the whole genomic additive genetic effect
- \( v \) is the vector of random effects of the regional genomic additive genetic effect
- \( \epsilon \) is the vector of residual errors

The variance for the vectors \( u, v \) and \( \epsilon \) are expressed by:

\[ u \sim (0, \sigma^2 G), v \sim (0, \sigma^2 Q), \epsilon \sim (0, \sigma^2 I), \]  

(2.6)

Where matrix \( G \) is a whole genomic relationship matrix using all SNPs for whole genomic additive effects, matrix \( I \) is an identity matrix for residuals and matrix \( Q \) is the regional genomic relationship (IBD) matrix corresponding to 100 adjacent SNPs that constitute the region of the genome analysed.
The phenotypic variance $\sigma^2_p$ is expressed as the sum of the whole genomic ($\sigma^2_u$), the regional genomic ($\sigma^2_v$) and the residual ($\sigma^2_e$) variances:

$$\sigma^2_p = \sigma^2_u + \sigma^2_v + \sigma^2_e$$  \hspace{1cm} (2.7)

The estimation of the whole genomic heritability ($h^2_u$) is expressed as:

$$h^2_u = \frac{\sigma^2_v}{\sigma^2_p}$$  \hspace{1cm} (2.8)

The estimation of the regional heritability ($h^2_v$) is expressed as:

$$h^2_v = \frac{\sigma^2_v}{\sigma^2_p}$$  \hspace{1cm} (2.9)

### 2.3 Software Implementation

#### 2.3.1 Suite of Programs

The current software implementation of the RGRM approach to analyse each region of the genome is compound by a suite of programs that are executed in the required order by a set of scripts. Some of these programs execute the core steps of the RGRM approach while others only manipulate the output data of a program to prepare the input data of the next program being executed. Table 2.1 lists the programs currently used and their role in the estimation of the genomic and regional heritability.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Program</th>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formation of the IBD matrix</td>
<td>script</td>
<td>R</td>
</tr>
<tr>
<td>Inversion of the IBD matrix</td>
<td>script</td>
<td>R</td>
</tr>
<tr>
<td>Data preparation and manipulation</td>
<td>scripts</td>
<td>Shell scripting</td>
</tr>
<tr>
<td>Estimation of the genomic and regional heritabilities by REML analysis</td>
<td>ASReml</td>
<td>Fortran 77</td>
</tr>
</tbody>
</table>

Table 2.1: List of programs involved in the analysis of each window of the genome.

The implementation currently allows to compute each region of the genome in parallel. For each region or window of the genome, an IBD matrix (the regional relationship matrix) is formed and then inverted. The formation and inversion of the IBD matrix are performed by a set of scripts written in R and shell scripts are used to manipulate and prepare the data during this process. Finally, the regional heritability for the particular region of the genome analysed is estimated by a REML analysis performed by ASReml. Figure 2.2 shows how these programs interact during the process.
ASReml is a serial program written in Fortran 77 which implements the AI REML algorithm to estimate variance parameters in general linear mixed models. It was originally developed by Arthur Gilmour and the program is currently developed and licensed by VSN International. The version of ASReml used to implement the RGRM approach is an unlicensed version and its source code is accessible. The computational efficiency of ASReml comes from its ability to handle very large data sets due to the use of the AI algorithm and sparse matrix techniques. One of the applications of ASReml are the genetic analysis of univariate and multivariate data but it is also applied for other purposes such as the spatial analysis of field experiments, the meta analysis of trials with common treatments and the repeated measures analysis.

2.3.2 Computing Challenges

Some initial insights and metrics regarding the computing efficiency of the implementation of the RGRM approach were initially outlined by the researchers. They elaborated their conclusion based on a genome-wide scan analysis for a single trait of 5511 overlapped windows of 100 SNPs (shifted every 50 SNPs) with data from a population of 3110 individuals. We categorized these conclusions into three different areas.

Memory Usage

The analysis of each window consumed up to 4 GB of memory. The geneticists desire to analyse larger populations using the RGRM approach, but it would imply to increase the memory requirements to run these analyses.
Execution Time

Based on the the same genome-wide scan analysis, the time required to analyse each window took around 30 minutes from which 20 to 25 minutes were took by the REML analysis performed by ASReml. With these numbers, the analysis of 5511 overlapped windows was estimated in around 2,500 hours of computing time. The geneticists argue that reducing the window size from 100 to 10 or 20 SNPs would increase their power to detect genes that affect the trait of interest but it would also mean to increase the number of windows analysed and therefore the computing time.

Use of Sparse Matrices by ASReml

The regional relationship (IBD) matrices formed for each region of the genome are mostly dense. The geneticists considered that the high computing time of the analysis could be explained by the usage of sparse matrix techniques by ASReml. The computations due to store the IBD matrix in sparse format and operating on elements of the matrix could possibly be avoided by modifying the implementation of ASReml to cope with dense matrices more efficiently.

2.3.3 Runtime Environment

The computer used to run the analysis of each window is a parallel Linux cluster called Eddie which is administered by the Edinburgh Compute and Data Facilities (ECDF). The current installation of Eddie offers two different types of computing nodes. Table 2.2 shows the specifications for each type of node and table 2.3 details the the CPUs installed on the standard nodes of Eddie. Besides the standard nodes, Eddie also offers a large SMP system with up to 480 GB of memory and 24 cores.

All the benchmarks described on this report were executed under this environment and any optimisation made to the implementation of the RGRM approach was targeted to this architecture. The Fortran compiler used was the Intel Fortran compiler (ifort) version 11.0.

<table>
<thead>
<tr>
<th>No. of Nodes</th>
<th>CPUs</th>
<th>Cores</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>2 Intel Xeon E5620 quad-core</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>156</td>
<td>2 Intel Xeon E5645 six-core</td>
<td>12</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 2.2: Computing nodes offered by the current installation of Eddie.
<table>
<thead>
<tr>
<th>CPU</th>
<th>Clock Speed</th>
<th>Cores</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>L3 Cache (Shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E5620</td>
<td>2.4 GHz</td>
<td>4</td>
<td>64 KB</td>
<td>256 KB</td>
<td>12 MB</td>
</tr>
<tr>
<td>Intel Xeon E5645</td>
<td>2.4 GHz</td>
<td>6</td>
<td>64 KB</td>
<td>256 KB</td>
<td>12 MB</td>
</tr>
</tbody>
</table>

Table 2.3: CPUs installed on the nodes of Eddie.

2.4 Optimisation of ASReml

As it was previously noted on this report, the REML analysis performed by ASReml accounted for 67 to 85 percent of the computing time taken to analyse each region of the genome. The improving of the computing efficiency of the RGRM approach and the scope for this project was narrowed to speed up the unlicensed version of the ASReml by parallelising the program using OpenMP. The reasons that supported the decision of using OpenMP are outlined on this section.

2.4.1 Incremental Parallelisation

Due to the fact that ASReml is a commercial program, we did not find any documentation about its implementation details. The code inspection of ASReml, described in Chapter 3, revealed that the source code was poor commented and with poor readability and maintainability. It led to dedicate a considerable amount of the time invested on this project to understand the implementation details of the program in order to attempt any parallelisation. Further analyses would also revealed a tight data dependency of the code. All these facts and the constraint of the time assigned to this project led to conclude that a radical change to the structure of the program was not feasible and that optimising ASReml by introducing incremental parallelism to the code was more suitable for this project. By using OpenMP directives we could be able to parallelise the critical routines of ASReml without the risk of breaking the rest of the code.

2.4.2 Serial and Parallel Versions

In addition to the benefit of implementing an incremental parallelism approach by using OpenMP, we also supported our proposed solution on the fact that the compiler directives-based approach of OpenMP would allow us to deliver a parallel version of the program as well as the original serial version using the same source code.

2.4.3 Loop-level Parallelism

An initial code profiling of the serial version (see Chapter 3) revealed that 88% of the execution time of ASReml was accounted by two routines of the program. These rou-
tines are iterative procedures implemented by loop structures that made them suitable to exploit a loop-level parallelism using OpenMP.

2.4.4 Parallelisation Within a Node

The analysis of each window of the genome was already computed in parallel by the implementation of the RGRM approach. It added support to the parallelisation of ASReml using OpenMP since we could perform the REML analysis of different windows in parallel on different nodes and exploiting a loop-level parallelism within a multicore computing node.

2.5 OpenMP

The OpenMP standard specification is extensive and it is not the purpose of this report to provide a deep explanation of every aspect. This section only details those aspects of OpenMP that were relevant to the parallelisation of the ASReml program. The code listings used on this and the rest of the chapters are written in Fortran but it should be relatively easy to port them to C or C++.

2.5.1 Programming Interface

OpenMP defines a portable programming interface for developing parallel applications and it has became the standard for programming shared memory multiprocessor architectures. OpenMP is not a programming language but a set of compiler directives that allow the programmer to describe the parallelism in the code, a library extension to existing programming languages and a set of environment variables. These three components define the OpenMP Application Programming Interface (API). There are implementations of the OpenMP API in Fortran, C and C++ and it is supported by all major platforms. The OpenMP standard is defined and revised by the OpenMP Architecture Review Board (ARB), a group of major hardware and software vendors, which approves the new versions of the specification.

Compiler Directives

The compiler directives defined by OpenMP are instructions to any compiler that supports it. These directives are identified by a sentinel at the beginning of the line of code. In Fortran the sentinel is defined as OMP and the compiler directives are expressed as source code comments. Fortran source code can be either fixed-form or free-form. This sentinel and any other OpenMP keyword are case-insensitive. If the parallel code is
compiled by a Fortran compiler that does not support OpenMP or the OpenMP directives are explicitly ignored using compilation flags then the OpenMP directives are just ignored by the compiler. This feature of OpenMP allows to write portable codes and produce serial and parallel versions of the program building the same source code.

Listing 2.1 shows how to express a simple parallelism in a Fortran code by using OpenMP directives. The simplicity of expressing parallelism by adding compiler directives to existing code is one of the major strengths of OpenMP. It provides to the programmer the ability of parallelising programs in an incremental fashion without affecting other parts of the code that are not intended to run in parallel.

Listing 2.1. Expressing parallelism with OpenMP directives

```fortran
program parallel_program

!$OMP PARALLEL
! a parallel region executed by a team of threads
!$OMP END PARALLEL

end program
```

Language Extension

The second component of the OpenMP API is a language extension implemented as new data type definitions and a library of routines. A OpenMP implementation provides a standard header file `omp.h` which provides type definitions and function prototypes exposed by the OpenMP API. This header file must be referred in any source code that depends on the OpenMP implementation. Table 2.4 list some of the Fortran routine prototypes included via the header file.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer omp_get_thread_num()</td>
<td>Returns the number of the calling thread within a team.</td>
</tr>
<tr>
<td>integer omp_get_num_threads()</td>
<td>Returns the number of threads in the current parallel region.</td>
</tr>
</tbody>
</table>

Table 2.4: Fortran routine prototypes exposed by `omp.h`.

Environment Variables

The final component of the OpenMP API is a set of environment variables meaningful to the OpenMP runtime environment. These environment variables provides a mechanism for controlling the parallel execution during the runtime, Table 2.5 list some of them.
The default values of these variables are implementation dependent. The value of these variables are read at the beginning of the execution of the program and any subsequent modification to their values is ignored.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS</td>
<td>Set the number of threads of execution inside a parallel region (team of threads).</td>
</tr>
<tr>
<td>OMP_SCHEDULE</td>
<td>Determine the schedule type for a parallel loop with a runtime schedule.</td>
</tr>
</tbody>
</table>

Table 2.5: Some environment variables meaningful to the OpenMP runtime.

### 2.5.2 Loop-level Parallelism

Exploiting parallelism in loops is one of the major strengths of OpenMP and one of its most common uses. The workload of a parallel loop is distributed among threads by assigning chunks of iterations to each thread. Figure Listing 2.2 shows how a simple loop is parallelised using OpenMP directives. `PRIVATE` and `SHARED` are some the clauses to express how the data among threads is shared inside the parallel loop.

**Listing 2.2.** Expressing loop level parallelism with OpenMP

```c
!$OMP PARALLEL DO PRIVATE (i) SHARED (matrix)
DO i = 1, 100
  matrix (i) = matrix (i) * 10
EN DO
!$OMP END PARALLEL DO
```

### 2.5.3 Loop Scheduling

One of the key factors that affect performance when we parallelise a loop using OpenMP is the loop scheduling. Loop scheduling defines how the iterations of a parallel loop are distributed among the threads. We defined the loop scheduling using the clause:

`SCHEDULE(type[, chunksize])`

**STATIC SCHEDULE**

When no chunk size specified, each thread is assigned with one chunk of iterations of nearly equal size. The chunks are assigned in order to each thread. If a chunk size is specified, the iterations are divided in chunks of size chunksize. Each chunk is then
assigned to each thread in a cyclic or round-robin fashion until no more chunks are remaining. When the number of iterations is not divisible by the number of threads, it is implementation dependent how the remaining iterations are assigned to the threads.

**DYNAMIC SCHEDULE**

Loop iterations are divided into chunks of size chunksize. If no chunk size is specified, the chunk size is 1. The list of chunks are assigned dynamically to threads at runtime. When a thread has finished with a chunk assigned, then it receives a new chunk until no chunks are remaining.

**GUIDED SCHEDULE**

Chunks are assigned dynamically to threads. The size of the first chunk assigned is large and implementation dependent. The size of each next chunk decreases exponentially being chunksize the minimum chunk size. If no chunk size is specified, the minimum chunk size is 1.

**AUTO SCHEDULE**

The decision on how to assign the iterations to each thread is decided at runtime.
Chapter 3

ASReml

3.1 Input Data

We were provided with a set of files for running the experiments with ASReml. This input correspond to the real data gathered from the genome analysis using the RGRM approach for serum and heigh traits from a population of 3110 individuals. The variance model of the genetic analysis implemented by ASReml expects a pedigree data file with the information of the individuals linked via a pedigree. Our pedigree file contains the information of sex, population, village and age of the population genotyped. By default ASReml will construct the covariance additive genetic relationship matrix (the correlation expected from the additive genetic effects) using the pedigree information and the inverse form of this matrix is the actual used by the program. When the inverse relationship matrix built by ASReml is not required, such is the case of the RGRM approach, the program takes a general inverse variance (GIV) matrix provided by the user. These files have a .giv file extension. The GIVs defined in our data set are the genome.giv file containing the genomic relationship matrix data and the region.giv containing the regional relationship matrix data of 100 adjacent SNPs. It is also possible to pass these files with a .grm extension in which case ASReml would invert the matrices. The data of the .giv files are expressed in a sparse format file. They define lines of three numbers (row column value) that represent the lower triangle of the matrix. The files are sorted by columns within rows. Finally, our set of files contain a command file with a .as extension. This file is used to describe to ASReml our pedigree data, specify our GIV files and our variance model. Table 3.1 summarises our input files and their purpose.

3.2 Mapping of Algorithm to Source Code

The first major task carried on was to map the iterative Average Information REML algorithm to the source code in order to have a deep understanding of the implementation details of ASReml. This was prerequisite before attempting any parallelisation of the
Table 3.1: Input data files used to run our experiments with ASReml.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pedASReml.txt</td>
<td>Pedigree information of population analysed (sex, population, village and age)</td>
</tr>
<tr>
<td>genome.giv</td>
<td>Genomic relationship matrix (all genome-wide SNPs)</td>
</tr>
<tr>
<td>region.giv</td>
<td>Regional relationship (IBD) matrix corresponding to 100 adjacent SNPs (window size)</td>
</tr>
<tr>
<td>asreml.as</td>
<td>Command file.</td>
</tr>
</tbody>
</table>

program. The estimation of variance parameter in linear mixed models by AI REML is an iterative process that stops until a convergence criteria is reached. First, we required to identify the Fortran procedures that were part of the AI REML iteration from those that were not. It allowed us to narrow our research on the critical sections of the code and the relevant memory data structures. Due to the lack of any available official or unofficial documentation about the implementation details of ASReml, the little or poor source code documentation and the unavailability of a code expert of ASReml, we faced a considerable delay on the progression of the project at this stage at it took considerably more time than the originally estimated.

3.2.1 Code Inspection

ASReml is written in Fortran 77 and therefore it inherits all the bad programming practices promoted by its lack of some features that are present in modern programming languages or newer specifications of Fortran. It makes ASReml a program with poor code readability and maintainability which is hard to modify. We describe here some issues faced for understanding the code implementation. Firstly, Fortran 77 restricts the maximum length of variable and procedure names to six characters and all names of procedures in ASReml have names such as G5VXXA,G5VWA,G5VVDG or G5VVVG which in addition to the lack of any proper source code documentation makes it almost impossible to decode their purpose. The same apply for names given to variables. Listing 3.1 is an extract of code from the G5VITR routine that shows how the code readability is decreased by the same reasons stated before.

Listing 3.1. Poor code readability. Extract of G5VITR subroutine

```fortran
MAXWVS=(IX (2,LXA+KEYABS+NEQD−1)−MAX(NEQ,NROW))/2
IF (MAXWVS.LT.3*MAX(NEQ,NROW)) CALL G5VFLT(IFAULT,6,*9000)
LBWV=LSSPS
LWK=LBWV+MAXWVS*2
LWKI=LXA+MAXWVS*2
LENWVS=NROW
```

16
The use of \texttt{GO TO} sentences allowed by Fortran 77, is considered now a bad programming practice. The ASReml code is full of them and it makes difficult to follow up the control flow of the program. They do not only harm the code maintainability but we also considered that they were maybe possible inhibitors of some compiler optimisation. Listing 3.2 shows shows an example of these bad uses of \texttt{GO TO}; In the other hand listing 3.3 shows a valid use of it in ASReml. Fortran 77 does not provide a \texttt{DO WHILE} control structure so \texttt{GO TO} is a workaround to this limitation.

\textbf{Listing 3.2.} Bad programming practice: Use of \texttt{GO TO} sentences. Extract of G5VWA subroutine.

\begin{verbatim}
IF (LBEQ(1, LBEQ(2, EQN)).LT.0) GO TO 200
IF (IXDWW(1, EQN).EQ.0) GO TO 200
I=EQN
J=1
100 KK=IXDWW(1, I)
105 IF (J.LT.KK) THEN
    WR(J)=ZERO
    J=J+1
IF (J.GT.NWV) GO TO 125
GO TO 105
END IF
\end{verbatim}

\textbf{Listing 3.3.} GO TO used to implement a \texttt{DO WHILE} control structure. G5VITR subroutine prototype

\begin{verbatim}
10 II=II+1
IF (KYXZ(II).GT.1) THEN
    IF (IDGGIU(1, KYXZ(II)).EQ.0) GO TO 10
    ID=KYXZ(II)
    IP=NR
15 CALL G5VWVS(WV, IWV, LENWVS, MAXWVS, IP, IDGGIU(1, ID),
        DGGIU(2, ID)*YXZ(II))
    ID=IDGGIU(2, ID)
IF (ID.GT.0) GO TO 15
GO TO 10
\end{verbatim}

Besides the code readability issues state before, we also confronted other bad programming practices that made the mapping of the algorithm harder. The core subroutines defined in ASReml are lengthy in the number of parameters. Listing 3.4 shows the prototype of the subroutine G5VITR. The routine takes more than fifty arguments. The
main issues with these type of declarations are that they decrease the program modularity and it results harder to spot the side-effects of the procedure since Fortran passes by reference all the variables to the subroutine call. In addition to this, implicit declarations of variables in Fortran 77 are allowed (resolved by using \texttt{IMPLICIT NONE} in later specifications). Identifying the purpose of the subroutines and their side-effects resulted in a long debugging process and it would also complicated the unit testing during the parallelisation of the program.

Listing 3.4. G5VITR subroutine prototype. Lengthy parameters declarations made hard to spot the side effects of the ASReml procedures.

\begin{verbatim}
SUBROUTINE G5VITR (NEQ, NROW, LYXZ, YXZ, KYXZ, KEYD, NSPAT, NEQD, NSECT, NRSECT, NTRT, DPMV, NCODE, NAINV, LCAINV, AINV, LLAINV, NFACT, NLFACT, TYFACT, NSTRUC, STRUC, NGAMMA, GAMMA, PGAMMA, TGAMMA, GCNSTR, STEPSZ, LOGLIK, SOLN, VSOLN, RESID, SIGMA2, NEDF, AI, NWV, NOEFF, NSING, YSSQU, GEQUAL, GCON, LGMCON, NGMCON, LEX, X, IX, ITERNO, AILOPT, NAIOPT, NIDR, IFAULT, DET)
\end{verbatim}

3.2.2 Fortran Procedures of AI REML iteration

After a long code inspection process we were able to identify the Fortran procedures in ASReml that mapped to the steps involved on each iteration of the AI REML algorithm. Table 3.2 summarises the steps required by each iteration and the ASReml routines that implement them.

<table>
<thead>
<tr>
<th>Iteration Step</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Addition of the matrix $G^{-1}$</td>
<td>G5VXXG</td>
</tr>
<tr>
<td>2. The formation of the total sum of squares and products (SSP) matrix.</td>
<td>G5VXXS</td>
</tr>
<tr>
<td>3. Reordering of the equations for retaining a high level of sparsity.</td>
<td>G5VEOR</td>
</tr>
<tr>
<td>4. Absorption of the SSP matrix and backsolve of matrix for solutions and calculation of residuals.</td>
<td>G5VXXA</td>
</tr>
<tr>
<td>5. Formation of the working variables.</td>
<td>G5VWVG</td>
</tr>
<tr>
<td>6. Formation of cross products of working variables.</td>
<td>G5VWXV</td>
</tr>
<tr>
<td>7. Absorption of the working variables to obtain the AI matrix and inversion of the sparse inverse of the coefficient matrix.</td>
<td>G5VWVA</td>
</tr>
<tr>
<td>8. Calculation of the score for the variance components.</td>
<td>G5VSCG,G5VSCR</td>
</tr>
<tr>
<td>9. Update the variance parameters.</td>
<td>G5VUPD</td>
</tr>
</tbody>
</table>

Table 3.2: Steps of ASReml iteration controlled by G5VITR subroutine.
The reordering of the equations by step 2 is a conditional step. It is only executed by the first iteration if the number of equations formed is greater than the number of dense rows. The reordering can forced ASReml to skip the reordering by passing the argument ISORT 0 to program execution.

3.3 Sparse Matrices Storage

One of the key strengths of ASReml is its ability to work with large data sets. This is possible due to the use of sparse matrices techniques. A sparse matrix storage format is used to reduce the amount of memory required for storing large matrices with many zero values. This section describes the storage format and the routines that implement the core operations for reading, updating and inserting elements of the sparse matrix. We have created a small IBD relationship matrix in order to make easier to represent graphically the sparse storage format and the operations on elements of the matrix. With this IBD matrix, ASReml formed a sparse matrix of 22 equations (2 dense and 20 sparse rows) with 160 non zero elements.

3.3.1 Sparse Matrix Storage Format

The core memory structure used by ASReml to store the shape of the sparse matrix is a linked list implemented as an allocatable two dimensional array (IX) of 2 rows and N columns. We call this the sparse storage index array. A node in the list represents a non-zero column in a row of the matrix. A node has two attributes: its value which is the column number that the node represents in a row and a pointer to the next column (node) in the same row. Figure 3.1 shows how the IX array is represented in memory. The first dimension of the array defines the column number (row 1) and the address of the next column (row 2) of a node. The second dimension of the array defines each of the nodes. The value of N is determined at the beginning of the program execution and it must be large enough to hold the sparse matrix that it ASReml will form. The default value of N set by ASReml can be increased by passing the argument -S to the program execution. The total number of equations and the number of dense formed are stored in variables NEQ and NEQD respectively.

Figure 3.1: Linked list implemented as a two dimensional array (IX) to store the sparse matrix shape
In order to identify the columns corresponding to a row, ASReml performs some calculations. The address of the first non-zero column of a row \( EQN \) is the value of \( EQN - NEQD \). With the address of the first non-zero column we can obtain then all the non-zero columns of the row. Figure 3.2 shows the sparse storage index array formed for our sample matrix. Different colours identify the nodes that correspond to the rows 2, 12 and 22. It can be seen that obtaining the address of the first non-zero column, we can obtain then the rest of the row’s columns.

![Figure 3.2: Representation of the sparse storage index in ASReml](image)

Figure 3.3 shows the shape of the matrix formed by ASReml for our sample matrix. It can be seen that the matrix is sparse and the first two equations are dense. Dense equations are not stored in the sparse storage index and they receive a different treatment. By using sparse matrix techniques ASReml only required to store 160 elements of the matrix instead of 22 X 22.

### 3.4 Operations on Elements of the Sparse Satrix

Once that we have explained how the sparse matrix storage index is created by ASReml, we review the operations that manipulate elements of the sparse matrix and the ASReml routines that implement them.
3.4.1 READ an entire row and its Diagonal Element

Reading an entire row of the matrix and getting its diagonal element is implemented by the routine G5VXIL. The process is described by Figure 3.4. Since the number of non-zero elements of the row is unknown, the columns are retrieved in a descending order. The diagonal element of row is its last non-zero column). Once that a row has been read and it is not needed any more, the nodes of the sparse storage index are released so they can be reused.

3.4.2 READ the value of a row’s column

Obtaining a column’s row is implemented by the routine G5VXIJ. Figure 3.5 show the process to obtain the column 5 in the row 12 in our sample matrix. A sequential search of a linked list makes easier to represent the search of the sparse storage index. It can be seen that getting the column 5 required 5 memory loads. In terms of performance, the worst case scenario would be reading the last column of the row since it would require a search through the entire list.

3.4.3 UPDATE the value of a row’s column

The routine G5VLLI implements the update or insertion of a column in a row. When the column to be updated is already a non-zero column in the row, the column is searched on the sparse storage index array and its value is updated. Figure ?? shows how the value of column 5 in row 12 in our sample matrix would be updated.
Figure 3.4: Reading a row and its diagonal element by G5VXIL routine.

Figure 3.5: Reading the value of a row’s column by GVXIJ routine.
### 3.4.4 INSERT a column’s row

G5VLLI routine also implements the insertion of columns in a row. If the column to be updated is a zero column in the row (not present in the sparse storage index), then a new node is inserted in the list in the right position. Figure ?? shows the process of inserting a new row’s column in our example.

### 3.5 Profiling of Serial Version

We profiled the serial version of ASReml in order to identify the Fortran procedures that were taking most of the computing time of the program. Then we narrowed our profiling to the steps involved on each iteration. Since the use of sparse matrices techniques by ASReml we also were interested in measuring cache-related metrics. The experiments were based on the input data described in section X and the program was executed on the backend of a standard node of the Eddie parallel cluster.

#### 3.5.1 Experimental Results

Identification of computing demanding routines

Figure 3.8 shows a flat profiling that measured the execution time that each ASReml routine took alone. It was interesting to note the notable optimisation job done by the compiler when running with -O3. ASReml is a code dominated by loop control structures so the many opportunities of loop optimisations could explain the dramatic increase in performance of the optimised version. I also dismissed our original concern about the use of GO TO sentences as inhibitors of compiler-based optimisations. Based
on this results, we determined the fraction of the total execution time taken by each subroutine alone. Figure 3.9 shows that four of the routines accounted for almost 90% of the total execution time. G5VXXA and G5VXXA subroutines implement the steps 4 and 9 of the algorithm iteration (see Table 3.2) and G5VLLI subroutine and G5XIJ function are associated with the insertion/update and retrieval of an element of the sparse matrix, respectively. It is interesting to note that with the optimised version G5VLLI and G5VXIJ increased their fraction taken of the total execution time while G5VXXA and G5VWVA decrease them. It seemed to confirm that the main performance issue of ASReml was due to the overhead of calculating indexes to access elements of the sparse matrix and to the disperse access of memory locations. All subsequent analysis consider the optimised version (-O3) of the program.

We performed then a call-graph profiling to figure out how many of the calls made to G5VLLI and G5VXIJ took part inside G5VXXA and G5VWVA. Table 3.3 shows that almost 100% of the calls were made by both subroutines. Finally, using these data we estimated the percentage of total execution time taken by all routines. Figure 3.10 shows the fraction of time that is taken by G5VLLI and G5VXIJ inside G5VXXA and G5VWVA routines and the time taken by being called by other subroutines of the programs. We concluded then that G5VXXA and G5VWVA subroutines, which accounted for 88% of the execution time, were the target of optimisation for this project.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Total Calls</th>
<th>Called By</th>
<th>Caller</th>
<th>% of Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>G5VLLI</td>
<td>2,272'040,296</td>
<td>2,197'373,768</td>
<td>G5VXXA</td>
<td>96.71</td>
</tr>
<tr>
<td>G5VXIJ</td>
<td>2,183'048,432</td>
<td>2,153'158,004</td>
<td>G5VWVA</td>
<td>98.63</td>
</tr>
</tbody>
</table>

Table 3.3: Calls made by G5VXXA and G5VWVA subroutines to G5VLLI and G5VXIJ.
Figure 3.8: Fraction of total execution per routine with -O0 and -O3 compiler optimisation.

Figure 3.9: Fraction of total execution per routine with -O0 and -O3 compiler optimisation.
Figure 3.10: Consolidation of fractions of total execution times taken by ASReml subroutines.

Analysis of ASReml iterations

Having identified that G5VXXA and G5VWVA were the most computing demanding subroutines, we timed the execution of each ASReml iteration. The purpose was to confirm that the iterations performed similar workloads and that time taken by G5VXXA and G5VWVA was distributed evenly among the iterations. We timed the ASReml iterations twice. As it was analysed in Chapter 3, the third step of the algorithm (see Table 3.2) is a conditional step that reorders the rows of the matrix in order to retain a high level of sparsity and it is only executed by iteration 1. Since the relationship (IBD) matrices are mostly dense, we also experimented by forcing to skip the reordering of the matrix.

We analysed first the execution that performed the reordering of the matrix. It can be seen in Figure 3.11 that all iterations, except iteration 1, performed a similar workload, as expected. We identified that the difference in execution time between iteration 1 and the rest was due to both, the set up of the matrix and the its reordering. We narrowed our analysis to identify the time taken by each step of the algorithm. Figure 3.12 shows a comparison between the times of iteration 1 and the average times of iterations 2 to 6. It can be seen that the G5VEOR subroutine, which performs the reordering, was only executed by iteration 1 and that G5VXXA and G5VWVA subroutines took most of the computing time, as we expected.

The most important fact revealed by the analysis of each ASReml iteration was that skipping the reordering of the matrix reduced the execution time not only of the iteration 1, as expected, but the overall execution time taken per iteration. In order to identify the algorithm steps impacted by the reordering, we compared the average times of by each subroutine (discarding G5VEOR) of the algorithm in iterations 1 to 6 with and without the matrix reordering. Figure 3.13 shows that G5VXXA and G5VWVA subroutines...
Figure 3.11: Timing of ASReml iterations with/without reordering of the matrix.

Figure 3.12: Timing of ASReml iterations with/without reordering of the matrix.
Figure 3.13: Average times of iterations 2 to 6 taken by iteration steps with/without matrix reordering

were the subroutines impacted by the reordering. We concluded that reordering the IBD matrix (mostly dense) to maintain a high level of sparsity of the matrix was the source of the decrease in performance per iteration.

3.5.2 Discussion of experimental results

Based on the results obtained by profiling the serial version of ASReml, we concluded that the Fortran compiler made an outstanding optimisation job using a -O3 optimisation level. We identified that subroutines G5VWVA and G5VXXA took together approximately 88% of the total execution time of ASReml (60 and 28% respectively). They implement the steps 4 and 7 (see 3.2) of the iterative AI REML algorithm performing intensive matrix operations. It explained the reason why routines associated with operations on elements of the sparse matrix were called by G5VXXA and G5VWVA so many times. We also noted that calls to G5VLLI accounted for more than 40% of the time taken by G5VXXA per iteration. In the same sense calls to G5VXIJ accounted for 17% of the time taken by G5VWVA per iteration. Finally, we concluded that time per iteration and the overall execution time of ASReml were notably reduced by skipping the reordering of the matrix step due to the fact that our IBD matrices are mostly dense.

All these findings outlined our two areas of optimisation for ASReml: the parallelisation using OpenMP of matrix operations in G5VXXA and G5VWVA and the rework of G5VLLI and G5VXII routines in order to improve the performance of operations on elements of the sparse matrix.
Chapter 4

Parallelisation of ASReml using OpenMP

4.1 Revision of Memory Data Structures

As we have explained on this report before, the computing efficiency of ASReml relies on the use of sparse matrices. The sparse storage index array (IX) works as a linked list implemented using a static two-dimensional array. This implementation presents two important issues. First, the waste of memory due to the static nature of the array is notable. Contrary to a linked list implemented using pointers and dynamic allocation of nodes, the length of the array remains the same during the entire execution of the program. The second issues is related to data locality. The disperse access to memory caused by accessing the columns of a row is not cache-friendly.

We thought that the first issue could be resolved by implementing a dynamic linked list data structure; however, it would present the same performance issues in terms of data locality as the array-based implementations. Also, the complexity implied in reworking the memory data structure could not be afford on this project due to the time constraint. We concluded that the data locality issue was more important in order to achieve a better performance and therefore we figured out a way to use a new local data structure inside the critical routines G5VWVA and G5VXXA.

4.2 Improve of code readability and maintainability

As we noted on Chapter 3, the poor readability and maintainability of ASReml made harder to modify the program. Before parallelising the code using OpenMP, we decided to refactorise the routines G5VXXA and G5WVVA in order to provide a cleaner code that facilitated further modifications and our implementation of OpenMP. These changes are described later on this section.
4.3 Description of Changes

G5VXXA Routine

The G5VXXA routine was broken into two new different source code files. Listings 4.1 4.2 describes the new routines defined to absorb the dense and sparse equations of the sparse SSP matrix.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSORB_SPARSE</td>
<td>Absorbs sparse rows of the matrix in parallel using OpenMP. Intermediate values are stored in a routine-local data structure.</td>
</tr>
<tr>
<td>READ_SPARSE_MATRIX</td>
<td>Reads the global sparse matrix into a local data structure in order to improve the memory access pattern.</td>
</tr>
<tr>
<td>STORE_ABSORBED_MATRIX</td>
<td>Copies the absorbed sparse matrix from the local data structure to the global sparse matrix.</td>
</tr>
<tr>
<td>BACKSOLVE_SPARSE</td>
<td>Backsolves sparse equations of sparse matrix.</td>
</tr>
</tbody>
</table>

Table 4.1: Routines defined in g5vxxa_sparsefinal.f source code file.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSORB_DENSE</td>
<td>Absorbs dense rows of the sparse matrix.</td>
</tr>
<tr>
<td>BACKSOLVE_DENSE</td>
<td>Backsolve dense equation of the sparse matrix.</td>
</tr>
</tbody>
</table>

Table 4.2: Routines defined in g5vxxa_densefinal.f source code file.

The critical part of the G5VXXA routine was the absorption of the sparse equations of the sparse matrix. Introducing a new data structure local to the routine we were able to remove the calls to G5VLLI subroutine in order to parallelise the main loop using OpenMP. Listing 4.1 shows the parallelisation of the main loop in ABSORB_SPARSE subroutine. As it is later referred on this report, DYNAMIC,16 was the best loop scheduling and chunk size for this loop.

Listing 4.1. Parallelisation of absorption of AI matrix using OpenMP

```c
C$OMP PARALLEL DO SHARED(matrix, x), DEFAULT(PRIVATE), C$OMP& FIRSTPRIVATE(eqn, xx, neqd, nrow, rcol, i) SCHEDULE(DYNAMIC, 16)
DO j = 1, i
   val = -(matrix(j, eqn)) * xx
   rrow = j
```

30
IF((eqn.GT.neqd + nrow).AND.(j.GT.neqd)) THEN
  IF(rcol.GT.neqd + nrow) THEN
    rrow = eqn - rcol + j
  ELSE
    rrow = j + (eqn - nrow) - (neqd + 1)
  END IF
ENDIF
ENDIF

IF(rcol.LE.neqd) THEN
  ii = rcol * (rcol - 1) / 2
  x(ii + rrow) = x(ii + rrow) + val
ELSE
  matrix(rrow,rcol) = matrix(rrow,rcol) + val
ENDIF
END DO
C$OMP END PARALLEL DO

G5WVVA Routine

In the same way that G5VXXA subroutine, we broke down subroutine G5VWVA into two different source code files in order to improve the code maintainability. Listing 4.3 and 4.4 summarises the subroutines contained on each file. The most computing part of step of the algorithm implemented by G5VWVVA routine was the inversion of the Average Information (AI) matrix. The multiple calls to G5VXIJ in order to read the values of the AI matrix were removed due to the use of a new local data structure; however, the main loop performing the inversion process could not be parallelised using OpenMP due to irremovable data dependencies. Nevertheless, the overall computing efficiency of the loop was improved by discarding the costly costs of calling the G5VXIJ routine.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI_ABS_SPARSE</td>
<td>Performs absorbtion of sparse equations of the Average Information matrix.</td>
</tr>
<tr>
<td>AI_INV_SPARSE</td>
<td>Invert the sparse equations of the Average Information matrix.</td>
</tr>
</tbody>
</table>

Table 4.3: Routines defined in g5vwva_sparsefinal.f source code file.
<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI_ABS_DENSE</td>
<td>Absorbs dense equations of the Average Information matrix.</td>
</tr>
<tr>
<td>AI_INV_DENSE</td>
<td>Inverst the sparse equations of the Average Information matrix.</td>
</tr>
</tbody>
</table>

Table 4.4: Routines defined in g5vwva_densefinal.f source code file.
Bibliography


