Optimising PLINK

Weronika Filinger

September 2, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

Every year the amount of genetic data increases greatly, creating the need for the tool capable of analysing large data sets in a fast and efficient manner. One such software package, providing a wide range of functionality required in whole-genome association studies is PLINK. Although, it does not limit the size of the data sets, the time needed to process them is often a bottleneck. This master project was focused on improving the performance of two functionality options: epistasis analysis and haplotype blocks estimation. It has been determined that the *g++* compiler and *–O2* flag provide the optimal performance for both options. The epistasis analysis has been parallelised using OpenMP. The *parallel for schedule* directive has been used and dynamic schedule with the *chunksize* of the size 128 provided the best scaling. When executed on 12 threads the epistasis analysis was 10.5 times faster than when executed on 1 thread. Haplotype blocks option has been serially optimised. Introduced optimisations improved the execution time by about 30%.

Contents

List of Tables

List of Figures

List of Extracts

Acknowledgements

I would like to thank: my supervisor Dr Alan Gray (EPCC) for guidance and advice, Dr Mairead Bermingham from the MRC University Unit for Human Genetics for providing the genetic data and answering all of my genetic-related questions, and others supporting me silently and patiently bearing my crankiness. Thank you.

Chapter 1

Introduction

With the advance in the fields of genetics and genomics the amount of available genetic data increases greatly every year. To identify the genetic regions responsible for physical traits like diseases, it is necessary to analyse large quantities of data from large number of individuals. Moreover, to reduce the risk of obtaining numerous associations by chance (false positive results), it is necessary to repeat analyses multiple times. This creates the need for an efficient and effective tool that would provide wide range of functionality to allow comprehensive and fast way to analyse the genetic data. PLINK [1], being one of such tools, is a software package written in C++ and providing a wide range of functionality required in the genome-wide association studies.

In this project, two PLINK options have been optimised – Epistasis and Haplotype blocks analyses. The Epistasis option has been chosen because it is extremely computationally demanding and therefore, it would benefit greatly from the optimisation. The main approach to improving the performance of this function was parallelisation. Similar work has been attempted by the *InSilico Research Group* [2]; some inconsistency in the results of that parallelisation have been discovered and investigated.

The Haplotype blocks option has been chosen as a candidate for optimisation because it is important aspect of the research conducted by the collaborating scientist Dr Mairead Bermingham from the Roslin Institute. Both serial optimisation and parallelisation have been attempted but due to the code structure, serial optimisation became the main focus.

In chapter 2, we present the biological context of the genome-wide association studies, PLINK and its options that were focus of this project. The last part of this chapter is dedicated to the tools and resources used throughout the project. Chapter 3 is about profiling PLINK. We present the performance of four different options. Two of them are simple summary statistics analyses that were investigated to provide insight in the behaviour of different PLINK options executed on different sample sizes and under different settings. In the second part of the chapter the performance of the Epistasis and Haplotype blocks options are discussed, along with the proposed approach to optimisations. All of the profiling has been done with the default PLINK settings. Chapter 4 presents the results of the investigation of different compiler flags and compilers on the performance of both Epistasis and Haplotype blocks options. The focus of chapter 5 is the Epistasis option - code structure, parallelisation, correctness, scalability, and finally serial optimisations are discussed. Chapter 6 is dedicated to the Haplotype blocks option. First, code structure, then the serial optimisation of the most dominant functions and finally the parallelisation are discussed. The last chapter contains the summary of the work done, conclusions and suggestions for future work.

Chapter 2

Background Theory

PLINK is a software tool used to analyse the genetic data. Hence, the first subsection of this chapter gives a brief introduction into the modern genetic studies, ideas and motivation behind the whole-genome association studies and some terminology required to explain the basic PLINK functionality. In the following section, the main domains of the program functionality are described. Then we give few examples of genetic studies that have used PLINK software package. Finally, the last subsection describes tools and resources used throughout the project.

2.1 Genome wide association studies

In the past two decades great effort has been put into decoding and understanding human genome. Projects such as the *Human Genome Project* [3] and the *HapMap Project* [4] were focused on identifying all of the genetic markers. The results obtained through the Human Genome Project has shown that 99.9% of the 3 billion base pairs of genetic letters are identical in every person. It means that this 0.01% contains some crucial genetic information responsible for most common, complex human traits and diseases. This difference in the genetic makeup is the reason behind the physical difference between people, for example blood group, height and skin colour etc. This genetic variation is also the reason why some people inherit rare diseases (e.g. muscular dystrophy or cystic fibrosis) and other are more likely to develop common illnesses (e.g. asthma, heart disease or cancer). Clearly, understanding the structure of the DNA not only provides important information on human evolutionary history but also is necessary to develop a new approach towards many diseases and their treatment. Being able to identify the genetic markers that influence individual health, would allow creating more precise and of reduced risk treatments.

Before discussing how the associations between the genetic variation and particular diseases are investigated, it is necessary to have basic understanding of how genetic information is inscribed. Genetic variation occur through mutations in deoxyribonucleic acid (DNA) which is composed from four basic building blocks molecules (bases) A,T,C and G. The order in which they occur is incredibly important for all life and body functions. The change of one base for another at single location is called single nucleotide polymorphism (SNP). Usually, a SNP does not have biological importance, sometimes however it can change the function of the gene. It is believed that there are about 7 million SNPs in human genome. Most common SNPs have only two alleles (alternative forms of a gene), the one at lower frequency is referred to as minor and the other as major. The common SNPs, those with the minor-allele frequency greater than 10%, occur approximately once every six-hundred bases [5,6]. Venter et al. [6] showed that the average human gene is 2,700 bases long which means

that about 50 common polymorphisms may be present in such gene. Moreover, the common variants tend to cluster into local neighbourhoods – haplotypes, which are groups of statistically associated SNPs present on a single chromosome. This characteristic allows the prohibitively expensive analysis of genetic variants to be reduced to analysis of carefully chosen SNPs.

Until recently, only the regions suspected of being involved in the development of a certain trait (usually disease) were investigated in the relation to that trait. The candidate gene proved to be unrelated more often than not and so finally new, unbiased approach has been developed - whole genome association studies (also referred to as genome wide association studies) [7]. In this approach, the analysis covers the whole genome when searching for the variation in human DNA underlying particular trait. In general, the whole genome association studies (WGAS) aim at identifying the genetic differences between the affected (cases) and unaffected (controls) by particular trait individuals. If a particular SNP is more common in an affected than in unaffected individuals, it is said to be associated with the investigated trait. The power and precision of the WGAS, and hence the ability to detect SNPs responsible for complex traits and diseases, are determined by the strength and the frequency of the linkage disequilibrium (LD) and trait modulating SNPs (directly affecting particular trait) [8].

Linkage disequilibrium is a measure of the correlation between alleles at two or more sites in the same region of the genome. In other words, when a certain combination of the genetic markers occurs in a population more or less often than would be expected from the frequencies of alleles forming the haplotype blocks, it is due to the linkage disequilibrium. This dependence between the SNPs may be formed when a novel SNP occurs in the region of fixed alleles where the other SNPs are present. This way the new SNP becomes fully correlated with the flaking haplotypes. Later this genomic region is transmitted through generations but as the DNA is breaking and re-joining at the germ cell development, different copies of original allele will end up with different LD patterns. They will also be flanked by different lengths of DNA from the original chromosome. Therefore, LD is generated by the transmission of these short genomic regions in unrelated individuals from the remote common ancestor.

Usually, in whole-genome association studies an additive framework is used to model genetic variation. In other words, it is assumed that each variant affecting the trait acts in an independent, linear and cumulative manner [9]. However, it seems that the genes action is more complex than that predicted by the additive model. One of the sources of non-additive genetic variation is epistasis. It is an interaction between two or more genes that affects phenotype (physical traits based on genetic and environmental influences). It is possible for genes to combine to produce a new trait or to mask each other's presence. In simple terms, epistasis occurs when an allele at one locus modifies or masks the effects of alleles at other loci, or when two or more loci interact to create new phenotypes. Epistasis provides important clues towards understanding many genetic and evolutionary processes. For example, evolution of sexual reproduction, speciation and the origin of life [10]

Two important aspects of the WGAS are genome coverage and the sample size. Typical WGAS covers at least hundreds of thousands of SNPs genotyped for thousands of individuals which makes them computationally challenging. In recent years many software packages have been written to provide the tools for genetic data analysis. Most of them are highly specialised i.e. have been written with specific analysis in mind (e.g

IMPUTE [11], Haploview [12]). One of the software packages offering a wide functionality is PLINK (one syllable). It is being used extensively in genome-wide association studies.

2.2 PLINK

PLINK [1] is a free, open-source C/C++ single-threaded, command line program designed to analyse the whole-genome data in a number of ways. It has been developed by Shuan Purcell at the Centre for Human Genetic Research (CHGR), Massachusetts General Hospital (MGH) and the Board Institute of Harvard & MIT, with the support of others. The PLINK has been released in September 2007 and the latest (possibly last) version – 1.07 – has been released in October 2009. The main purpose of PLINK is manipulating and analysing of large whole-genome data sets in their entirety. Its initial functionality covered five main domains: data management, summary statistic, population stratification, association analysis and identity-by-descent estimation. Multiple other features have been added in latter releases (e.g. epistasis). In the next few paragraphs, the main functionality domains and other 2 options will be described briefly.

As a part of data management functionality, it is possible to: read data in variety of formats (normal text files, binary files and transposed files), recode and reorder files, merge two or more files, filter genotype information (e.g. extracting subsets of SNPs or individuals) and change the data file format. Three main file formats are supported: text, transposed and binary. Normal, text file has a one-row-per-individual format, and the transposed file has one-row-per-SNP format. Finally, the binary file is a more compact way of representing SNP data. Typical PLINK function requires two or more data files to perform desired operation – *ped* and *map* file in text format, *tped* and *tfam* in transposed format and bed, *bim* and *fam* in binary format.

The *ped* file is a white-space delimited file, containing family and individual information in six mandatory and two optional columns. They mandatory columns contain family ID, individual ID, paternal ID, maternal ID, sex and phenotype, respectively. The two additional columns contain names of alleles. The map file, as the name suggest contains the genetic map and has four columns: chromosome number, SNP identifier, it's genetic distance (in morgans – unit for measuring the genetic linkage) and finally base-pair position (one kilobase equals to 1000 base pairs of DNA). The transposed file set holds the genotype information (one row per SNP) in *tped* files and individual and family information (one row per individual) in *tfam* files. All the information is represented in the same way as in *ped/map* file format, except the information is swapped between files. The transposed format is preferable when the data contains many more SNPs than individuals (often the case with WGAS data), this way the *tped* file will be long as opposed to the *ped* file being wide. The binary file format splits information into three files: the binary bed file (containing genotype information), the text file *fam* (storing phenotype information) and text file *bim*, which is an extended map file (contains two extra columns with allele names).

PLINK allows manipulation of the way those files are read in e.g. some of the variables can be read in in different units, some columns/rows may be skipped and so on. There are also other file formats available which are either mixed forms of above formats (e.g. long-format file) or contain more specific information required by some of the PLINK options (e.g. alternate phenotype files). To specify the file format and make use of extra options, additional command line flag needs to be added to the execution command.

The summary statistics is mostly used for quality control of analysed sample. Multiple summary measures are available and can be used to obtain general information about the data sets. In this report we will discuss briefly two functions belonging to the summary statistics domain: missing genotype rates and allele frequencies. Both are part of the PLINK tutorial available online [13], and as such were profiled to give better insight into the structure of the program.

PLINK provides a wide range of options to perform the association analysis, some of the test can be conducted in multiple ways (e.g. on different genetic models, with permutation etc.). There are case/control tests, family-based association tests, quantitative traits tests, tests for stratified samples and others. The most basic association test is based on comparing the allele frequencies between cases (affected individuals) and controls (unaffected individuals) to identify the regions responsible for a particular trait.

The next domain allows performing identity-by-descent and identity-by-state estimation. Identity-by-descent (IBD) refers to the case where two or more alleles are identical copies of the same ancestral allele. In other words, the two alleles are inferred as IBD if enough of their SNPs are the same. The IBD analysis allows discovering the unknown familial relationships, as well as detecting sample contaminations, pedigree errors, swaps and duplications. It is also possible to find specific segments shared between distant relatives. Two alleles are referred to as identical by state (IBS) if they have the enough SNPs that are the same even though they do not share common origin. In other words, they have the same DNA sequence but not because they were inherited from common source.

The population stratification is a difference in the allele frequencies between groups in a population. In other words, members of a given population are grouped together based on their allele frequencies. PLINK uses the pairwise IBS distance with other restrictions (like cluster size, phenotype criterion and so on) to perform agglomerative clustering and investigate the population's substructure. The obtained results can reduce and refine the size of the sample for the subsequent association test (by cluster specification).

There are also other features supported by PLINK like for example epistasis. In fact, PLINK provides two ways of performing epistasis analysis. The first one is invoked by the *--fast-epistasis* and the other by *--epistasis* command. The main difference between them is the method used for calculating the pairwise interactions between SNPs. As the name suggests, the fast epistasis option is significantly faster than the normal one. More details are provided in chapter 5.

Some of the other PLINK options include multi-marker tests, meta-analysis and imputation. They have been added to later releases of the program to provide comprehensive functionality for whole-genome association studies. Thanks to that, PLINK is being used in wide range of genomic and genetic studies such as identifying

loci responsible for traits like body mass index [14], negative symptoms in Schizophrenia [15] or for cannabis dependence [16].

The use of PLINK in research is usually accompanied by other tools and techniques and at the same time many PLINK options are used. For example, the research using WGAS to identify the genomic region responsible for the size of spine bone in humans [17] used both quantitative trait association tests and population stratification. Ge at al. [18] when studying influence of genetic variation on hepatitis C treatment used the following PLINK options: calculation of relatedness (IBD estimation), Hardy-Weinberg equilibrium analysis (one of the summary statistics tests), logistic regression and population stratification.

Two of investigated studies had used the epistasis analysis provided by PLINK. One of them was focused on investigating statistical epistasis in complex disorders [19] (both fast epistasis and epistasis methods available in PLINK have been used) and the other on finding susceptibility loci for Crohn's disease [20] (case-only epistasis function used).

Sometimes PLINK is used to prepare the sample for the next tests and methods. For example one of the investigated researches [21] required LD-clumped input to apply further techniques and PLINK has been used to produce it. The basic idea behind the LD-clumping is to use the empirical estimates of linkage disequilibrium between SNPs and to group them accordingly across one or more datasets or analyses.

The wide range of functionality, ease of use and good documentation are some of the reasons why PLINK is commonly used in the scientific community. As mentioned before, PLINK is capable of handling large data sets and the only factor limiting the size of the analysed samples is the time necessary to process them. Some of the program functions require looping over all of the SNPs and all of the individuals and so is time consuming. Consequently, some of the researches are significantly hindered by the slow computation process. To make PLINK even more functional, it is necessary to improve the program performance.

PLINK provides numerous features and optimising all of them is way beyond the scope of this project. Also because it is so widely used and for so many different purposes, it is hard to decided what part of the functionality would benefit the most from the performance optimisation. Therefore, in this project we were focused on improving the performance of two options - epistasis and haplotype blocks analysis. The epistasis analysis has been chosen because similar work has been already attempted by the *InSilico Research Group* [2] and analysing it as a part of initial investigation was supposed to provide insight into potential optimisations of other PLINK options. However, the parallelisation done by *InSilico Group* seems to be incomplete and produce inconsistent results. The brief investigation on that parallelisation is presented in chapter 5. The inconsistencies and the fact that the Epistasis analysis is very time consuming convinced us to make parallelisation of the Epistasis option part of this project. The Haplotype blocks option has been chosen because it is relevant to the research of collaborating scientist from the Roslin Institute [22] - Dr. Mairead Bermingham. More details on the both options and in what kind of scientific research they can be used will be provided in further chapters. Meantime, in the next section, tools and resources used throughout the project are described briefly.

2.3 Resources and tools

Most of the development work has been conducted on the compute component of Edinburgh Compute and Data Facility (ECDF) – cluster *Eddie*. It consists of 156 IBM dx260M3 iDataPlex servers and each of them has two Intel Xeon E5645 six-core processors. To provide better insight into the program behaviour on different architectures, EPCC training machine *Morar* has also been used (Intel(R) Xeon(R) CPU E5-2407 0 @ 2.20GHz). It consists of 128 cores, divided into two shared-memory boxes of 64 cores.

All of the work has been done on development version of PLINK v 1.07. The PLINK source code has 132 files. Two of them are text documents – COPYING (containing GNU general public license v2) and README containing basic information about the program. There are also executable *jar* file (gPLINK providing GUI for PLINK) and two files of the format required by PLINK *test.ped* and *test.map*. The actual program consists of 32 header files, 94 code files and *Makefile*. PLINK is capable of running on the following platforms: Unix/Linux, Windows, Mac and Solaris. The platform used should be specified in the *Makefile*. There are also additional compilation options available: R plug-ins, web-based version check, ensure 32-bit binary, link to *zlib* (compression library), link to LAPACK and force dynamic linking. Thanks to the R plug-ins it is possible to use R package for statistical computing. During this project the default options were enabled: R plug-ins, web-check, *zlib* library and forced dynamic linking. PLINK does not require installation. However, on different systems the location of libraries is different. Hence it was necessary to specify the correct path for the *zlib* library. On the *ecdf* cluster *Eddie* it was */lib64/libz.so.1*. The default compiler specified in a *Makefil*e is *g++* and its default flags are *-O3* and *-I*. The compiler and flags are discussed in more detail in chapter 4.

The data used for the PLINK tutorial [13] and development of parallelised epistasis analysis comes from the PLINK website – downloadable as a *zip* file *hapmap1.zip*. The data are split between three files *hapmap1.ped, hapmap1.map* and *hapmap1.phe* (phenotype information). The sample contains 83534 SNPs from randomly selected 89 Asian individuals from HapMap [4]. Thus the data is not in any way representative for any study design or disease model but is good enough to illustrate PLINK behaviour. The work on the haplotype blocks analysis was carried out on *Genoplink_20130205* transposed data (*Genoplink_20130205.tfam* and *Genoplink_20130205.tped*) provided by the collaborating scientist Dr Mairead Bermingham. The data come from three different population-representative cohorts from the Croatian population. The sample consists of 2357 individuals from which 960 have been recruited from villages of Vis and Komiza on the island of Vis (between 2003 and 2004), 897 from Korcula island (2007) and 500 from city of Split (between 2008 and 2009). The individuals and SNPs not meeting certain criteria were excluded from the analysis. The final sample contains 267912 markers that 2186 individuals (871 males and 1315 females) had in common.

To profile the performance of the tutorial cases, epistasis and haplotype blocks options, *gprof* has been used. During the course of the project some minor bugs in the code have been identified and fixed (discussed in chapter 4). To test the correctness of the optimisations and parallelisation, first Linux *diff* command has been used. In cases

when this command was not sufficient to decide whether two output files contain consistent data or not, specially written test have been used (in C and python). Those tests compare the contents of two files and return the difference between them. The files are recognised as being consistent even if the data are stored in different order.

To resolve the problems with memory allocation encountered when parallelising haplotype blocks function Valgrind's DRD [23] tool has been used. DRD is used for detecting errors in multithreaded C and C++ programs.

OpenMP has been chosen as a parallelisation method. The choice has been mainly dictated by the nature of the code, the language in which PLINK has been written and the platform on which it is being used. Using OpenMP allows parallelisation through introducing *#pragma omp* pre-processor directives and so major re-engineering of the code is not necessary. The changes in the code usually involve ensuring the correct scope of variables and introducing critical sections where needed. Also OpenMP does not require any special wrappers or bindings for parallelisation of C++ programs.

The code contains many nested loop regions (mostly over SNPs and/or individuals) which are good candidates for parallelisation using *#pragma omp parallel for* directive. Furthermore, some of the PLINK options, like haplotype blocks, have quite complex, split among many code files structure. Those options usually make use of functions primary belonging to other options, which makes the call tree rather nested. In such cases orphaning provided by OpenMP seems to be a good and straightforward approach to parallelisation.

Finally, according to our knowledge the scientists from the Roslin Institute at the University of Edinburgh run PLINK either on *ECDF* cluster *Eddie* or on the internal cluster of a very similar structure. Being shared memory machine, it allows execution OpenMP parallel jobs. To run them, it is necessary to specify in the job script the number of OpenMP threads (via *-pe OpenMP*) and initialise the environment module (*./etc/profile./modules.sh*). Example of the job script is shown in appendix A. The parallel environment accepts requests for up to 12 job slots. Hence, in this project the parallel jobs on *Eddie* are run on a maximum of 12 threads.

Throughout the project, we have used the PLINK source code to run all of the analyses. Normally, the researches using PLINK on *Eddie* do not use the source code (if they are not interested in changing the compiler settings or modifying the code) because *Eddie* has two version of PLINK installed (v 1.06 - default and 1.07). Therefore, to run desired analysis, one first need to load the PLINK module and then use the appropriate command.

All of the measurements have been repeated three times and the one with the shortest execution time has been taken as a representative for the investigated settings. The reasoning behind this choice is our interest in how fast the code can be executed, not how and to what extend other processes can hinder it.

In next chapter we describe briefly how to run PLINK, what kind of output it produces and discuss the profiles of two of the simple summary statistics options. Then we will proceed to discussing the performance of epistasis and haplotype blocks options with default PLINK settings.

Chapter 3

Profiling PLINK

One of the first tasks undertaken in this project was profiling of different aspect of PLINK functionality. Although, optimising all of the aspects of the program is beyond the scope of the project, understanding how the program works is crucial in optimisation process. This way, the parts of the code that would benefit the most from optimisation can be identified. In this chapter four different PLINK options are discussed. Two of them are part of the summary statistics domain, and the other two are epistasis and haplotype blocks option. Firstly, we will discuss briefly what they are doing and what they are used for, then how to perform them using PLINK and finally we will present their profiles. All of the presented profiles were obtained for the default PLINK settings (i.e. compiler *g++*, flag *-O3*).

3.1 Simple cases

To understand how the program works and what it is capable of, we went through PLINK tutorial available online [13]. All of the tutorial examples have been profiled but only two cases, having considerably longer execution times (still very short), are discussed here. They are missingness rate and allele frequencies statistics. Both analyses are considered separately and in more details than required but we believe it is a good way to illustrate how PLINK works. Both of them were executed on both artificially small (*hapmap1*) and consisting of real life data (*Genoplink_20130205*) samples. Profiles generated for both options and sample sizes are compared, giving insight into general behaviour of the program. First we will discuss the missingness rate option.

3.1.1 Missingness rate

Missingness rate analysis checks the rate of missing genotype information (SNPs) in an analysed sample. It is particularly useful for refining the sample in preparation for other tests. If a particular individual is missing significant number of SNPs or if particular SNP is absent in large number of individuals, they can be excluded in further analysis. In other words, it is good quality control tool.

As mentioned in chapter 2, PLINK is a command line tool and so specific commands are required to perform desired analysis. To perform missingness analysis the following command has been used:

```
./plink --bfile hapmap1 --missing --out miss_stat
```

This command runs the executable *plink* using the binary data files called *hapmap1.bim* and *hapmap1.fam* as an input files. Then it performs the calculation of

the missingness rate (invoked by --*missing* command) and outputs the results into the *miss_stat.imiss* and *miss_stat.lmiss* files. The log file is also generated automatically (*miss_stat.log*).

The log file provides the basic information on the examined sample. At the top of the file there is a banner containing the copyright information and PLINK version. Then information about the web-check for the newer version, the time the analysis is started at, options in effect, and the name and location of the input files are specified. The next few lines contain the information about the analysed sample – number of SNPs and individuals, as well as how many of them are missing phenotype, whether they are affected (case) or unaffected (control) and their gender. Then the output files are specified and total genotyping rate in individuals with non-missing phenotype is given. Finally, how many SNPs have failed the test and the analysis end time.

Usually if the sample contains pre-selected markers and individuals, the reported genotyping rate is high. In the case of data from hapmap1 sample (83534 SNPs and 89 individuals) the rate is 0.99441. Also no SNPs and individuals were removed. The values of the thresholds (both for excluding SNPs that are missing in many individuals and individuals with too much missing genotype data) can be regulated via command line options (--*mind* and --*geno*, respectively).

As mentioned before, the missingness analysis produces two output files *miss_stat.imiss* and *miss_stat.lmiss*. The former, stores the genotyping missingness by individual (what portion of SNPs each parson in the sample is missing) and the latter by SNPs i.e. how many people are missing a particular SNP. For the individuals the file contains 6 columns: family ID, individual ID, missing phenotype, number of missing SNPs, total number of SNPs included in this analysis and finally the portion of missing SNPs.

Extract 1 shows first six lines of the *stat_miss.imiss* file. The family ID is represented by the identifier consisting of letters and numbers. The individual ID is simply a number ordering the individual within their family. In the hapmap1 sample all of the individuals are founders or in other words are representing different families and so all of them have the same individual ID – 1. If the individual is missing phenotype, letter Y is placed in the third column. According to the log file, there are no individual with missing phenotype in the *hapmap1* sample and therefore, all of the rows have N in the third column. The number of missing SNPs is different for each individual and clearly can vary significantly. Because the no SNPs have been excluded from the analysis, the N_GENO column (total number of included SNPs) is the same for all of the individuals and equals number of SNPs in the sample. The last columns, the portion of missing SNPs is simply N_MISS/N_GENO.

F_ID	I_ID	MISS_PHENO	N_MISS	N_GENO	F_MISS
HCB181	1	N	671	83534	0.008033
HCB182	1	N	1156	83534	0.01384
HCB183	1	N	498	83534	0.005962
HCB184	1	N	412	83534	0.004932
HCB185	1	N	329	83534	0.003939
HCB186	1	N	1233	83534	0.01476
…					

Extract 1 First few lines of *stat_miss.imiss* file obtained as a result of missingness analysis on *hapmap1* sample.

The other file has 5 columns: SNP identifier, chromosome number, number of individuals missing this SNP, total number of individuals included and the proportion of sample missing for this SNP. The first few lines of this file are presented in extract 2. All of the first column entries have chromosome 1 because the analysis proceeds chromosome by chromosome. Looking at the third column, it is clear that some SNPs are present in all of the individuals (N_MISS = 0). Again, because no individuals have been excluded from the analysis, the value of N_GENO in all rows is the same and equal to the number of individuals in hapmap1 sample. The proportion of individuals missing particular SNP is calculated as N_MISS/N_GENO.

CHR	SNP	N_MISS	N_GENO	F_MISS
1	rs9729550	0	89	0
1	rs3813196	0	89	0
1	rs6704013	2	89	0.02247
1	rs307347	12	89	0.1348
1	rs9439440	2	89	0.02247
1	rs3128342	1	89	0.01124
...				

Extract 2 First few lines of the *stat_miss.imiss* file obtained as a result of missingness analysis on *hapmap1* sample.

The missingness analysis has been performed on two samples of different sizes and represented in different formats – *hapmap1* (83534 SNPs, binary) and *Genoplink_20130205* (267912 SNPs, transposed). Both samples have been profiled using *gprof*.

Figure 1 shows the distribution of the total execution time for the *hapmap1* sample. Only the functions taking more than the 1% of total time are present in the figure; the contribution of the rest is under the *other functions* label. The execution time was 0.59 seconds. The most dominant is the routine responsible for filtering the SNPs, it accounts for almost 60% of the execution time. The second most time consuming function (*CheckDupes*) checks sample for duplicated individuals and markers, and takes almost 12%. The third is function *Plink::readBinData*, responsible for reading binary data into the program consumes 10% of the total time. The vector class is taking almost 7% of the total execution time. *Locus* class is also taking almost 7%. The last noticeable contribution, taking about 5%, comes from the calls to the *std* library. Other functions combined together take only 0.02% and thus are not examined closely.

The execution time of the bigger sample (*Genoplink_20130205*) was 124.41 seconds and its profile looks slightly different (Fig.2) than that of the smaller sample. The most dominant routine, responsible for reading the transposed data in, takes over 65% of the total time. The second most time consuming routine is the *Plink::filterSNPs* function with contribution of over 34%. On the third position is the *std* class taking 0.1%. Both of them were present in the profile of the smaller sample as well. The other named functions, taking less than 0.01% each, are *checkDupes*, the *vector* class and *Locus* class. Clearly, there are only two significant contributions to the total execution time.

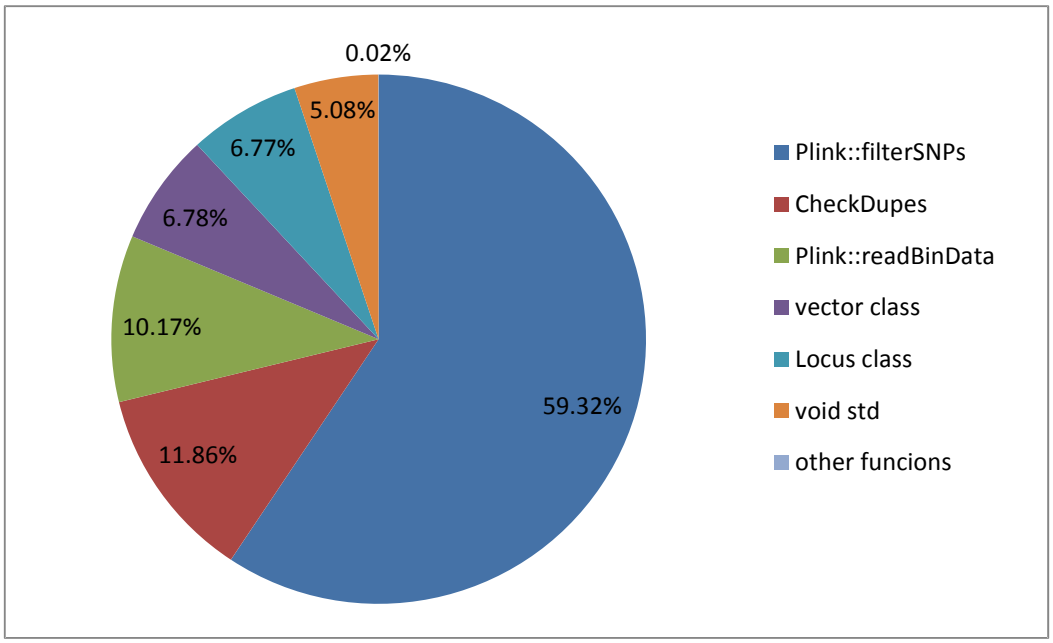

Figure 1 Profile generated for missingness analysis performed on *hapmap1* sample and with default PLINK settings.

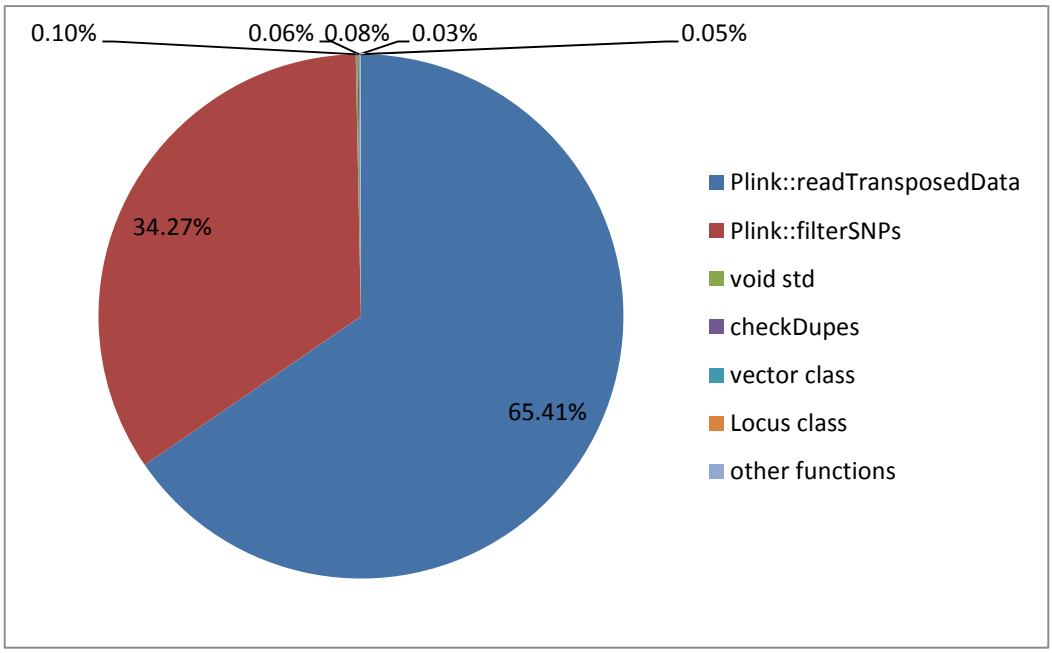

Figure 2 Profile generated for missingness analysis performed on *Genoplink_20130205* sample and with default PLINK settings.

Comparing both profiles, it is clear that with increasing size of the sample some of the functions become more and other less time consuming. Putting aside the time taken to read the data in (which will probably become less significant for more computationally expensive PLINK options) the contribution from the function responsible for performing the actual work, i.e. filtering SNPs, increased. Therefore, if bigger samples are considered (justified by WGAS characteristics) and if the computation is concentrated in small number of functions (characteristic of main PLINK options) the number of functions being the candidates for optimisation will be smaller. The contribution from the vector class has also become less noticeable in the profile of the bigger sample. The reason behind that may be that the vector class is actually not used that much in the computational part of the analysis. It is hard to tell how its contribution will look like if more complex PLINK option are performed.

Also it seems, that reading data in the transposed format (one row per SNP) takes longer time than the binary format. To investigate the effect of file formats on the execution time, the missingness analysis has been performed on the *hapmap1* sample represented in different ways. When the binary (*--bfile*) data file has been used (discussed above) the analysis took 0.59 seconds. Using the transposed file (*--tfile*) and normal text file (*--file*) resulted in analysis taking 2.16 and 3.91 seconds, respectively. Clearly, different file formats affect the performance of smaller samples. It is expected that this difference will become smaller when the analysis will be more computationally challenging. Although, the binary format is the fastest, the transposed seems to be more convenient when the size of the sample becomes large (which is typically the case in WGAS). Also the binary file cannot be read through simple viewing programs.

3.1.2 Allele frequencies

The second function, coming from PLINK tutorial, is responsible for calculating the frequencies of the alleles present in the sample. For each SNP present in a sample, it generates the list of major and minor allele frequencies. Again as a function from the summary statistic domain, it is used mainly for the quality control of the analysed sample. To perform the allele frequencies analysis the following command has been used:

```
./plink --bfile hapmap1 --freq --out freq_stat
```

PLINK first reads the *hapmap1* sample in binary format, then performs the allele frequencies analysis and finally writes the results to the *freq_stat.frq* file. The log file for this PLINK option contains essentially the same information as for missingness rate. The only exception is there is no information about removed SNPs as the analysis itself is not removing any SNPs but only producing the file with the allele frequencies.

The output file (*freq_stat.frq*) contains six columns. The first one stores the chromosome number and the second one the SNP identifier. The following two contain the allele code for minor and major alleles, respectively. The next one holds the minor allele frequency and the last one contains the non-missing allele count. The first six lines of the file are presented in extract 3. All of the entries in the first column are 1 because the analysis proceeds chromosome by chromosome. The allele codes are coded as 1 and 2 (0 if the allele is not present). The non-missing allele count is calculated as the difference between the number of individuals and the number of missing genotype records at the SNP.

CHR	SNP	A1	A2	MAF	NCHROBS
1	rs6681049	1	2	0.2135	178
1	rs4074137	1	2	0.07865	178
1	rs7540009	0	2	0	178
1	rs1891905	1	2	0.4045	178
1	rs9729550	1	2	0.1292	178
1	rs3813196	1	2	0.02809	178
...					

Extract 3 First few lines of the output file *freq_stat.frq* produced for *hapmap1* sample and with default PLINK settings.

Again, both the *hapmap1* and *Genoplink_20130205* data sets have been used, the total execution times were 0.32 and 97.49 seconds, respectively. Figures 3 and 4 show that both profiles show many similarities to the respective profiles obtained for missingness rates option. For the small, binary sample (hapmap1) the most dominant routines are again *Plink::filterSNPs* (41.18%), *Plink::readBinData* (20.59%) and *checkDupes* (14.71%). The order changed slightly (second and third functions are swapped) compared to missingness profile. On the fourth most time consuming position is Locus class (8.82%) followed by vector class (5.88%). The total contribution coming from the *std* class is also noticeable as it takes almost 9% of execution time. The same as in the case of missingness function the number of contributing functions is quite large. The other functions do not take enough time to be present in the profile.

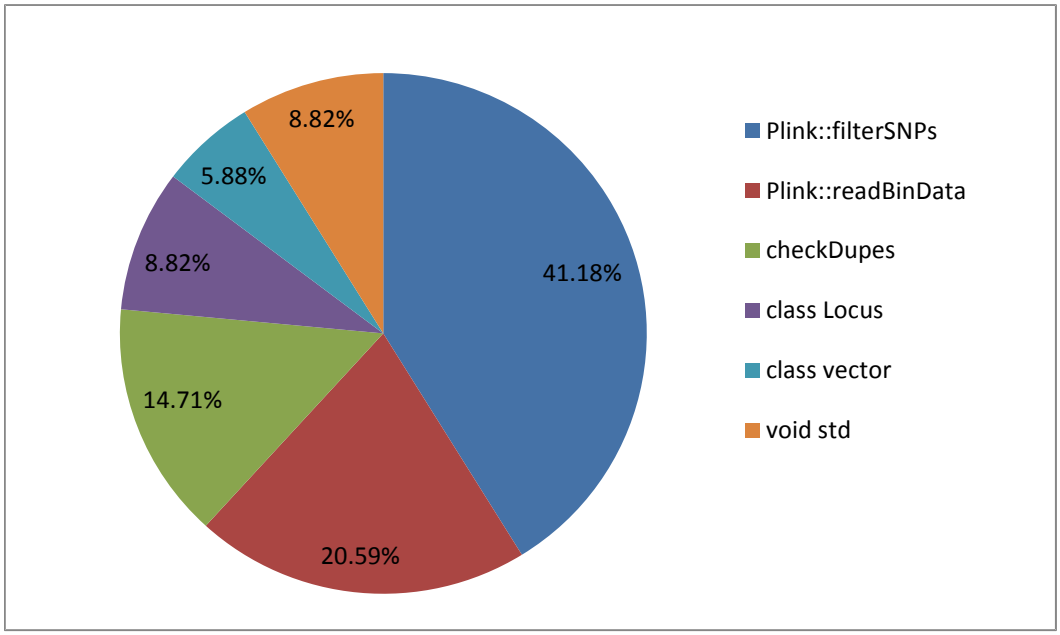

Figure 3 Profile generated for allele frequencies analysis performed on the *hapmap1* sample and with default PLINK settings.

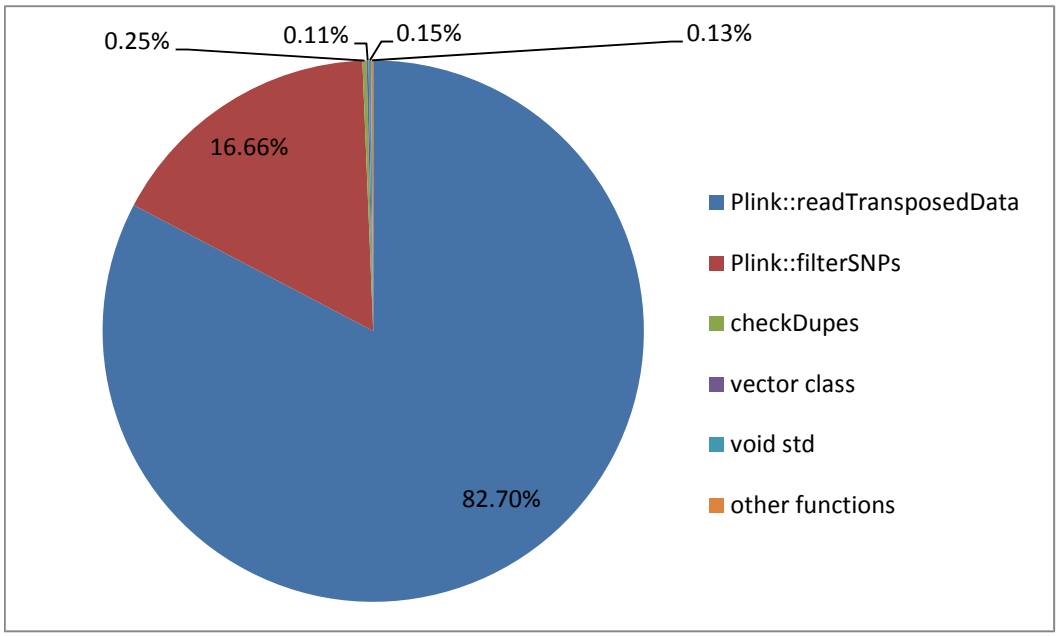

Figure 4 Profile generated for allele frequencies analysis performed on *Genoplink_20130502* and with default PLINK settings.

The tendencies for the bigger sample are also similar to those observed for missingness rates. The profile shown in figure 4 has the same dominant function as the one in figure 2. The most dominant is the function responsible for reading the transposed data in (82.7%), then the second is the function filtering SNPs (16.66%). Both of them account for almost entire execution time. The other functions named in the labels are again *checkDupes*, *vector* class and *std* class. Their contribution is smaller than for the missingness profile. The reason for that is most likely because the missingness rate analysis required sorting genotyping rates with respect to both SNPs and individuals, whereas the allele frequencies option sorts them only with respect to SNPs. In both cases, the time needed to read the data in is roughly the same for both PLINK options (~81s). The time spent on filtering SNPs is significantly shorter (done only with respect to SNPs) in the allele frequencies option.

3.1.3 Conclusions

Only two options form the PLINK tutorial have been discussed. Nevertheless, all of the analysed options exhibit similar behaviour for both datasets, irrespective of sample size. Although, for the bigger dataset the execution time was dominated by the time spent on reading the transposed data into the program, this behaviour is most likely typical for relatively small samples and computationally inexpensive options. Although, it is not part of this project, we believe that it may be advantageous if the future work was to improve the efficiency of the reading the data in, as the program is often used just for changing the format of the data. The contribution from the vector class differed for different sample sizes and PLINK options but it was present in all of them (not only those presented above). That is because PLINK stores most of the genetic data in dynamically allocated vectors. That is also the reason why other *std* class members are also present in the profiles – manipulating the data requires allocating, inserting, sorting, pushing back, resizing etc. Thus the vector class is used extensively across all of the program options.

3.2 Epistasis and Haplotype blocks option

In this part of the chapter we present the profiles of the fast epistasis and haplotype blocks option. Optimising them is the main focus of this project and therefore, they are discussed separately and in more detail than the tutorial cases. First, we will discuss briefly what are they doing and what they used for and then the profiles will be discussed. The fast epistasis option has been performed on slowest format of the *hapmap1* sample (normal text file) and for the haplotype blocks option *Genoplink_20130502* sample has been used.

3.2.1 Epistasis

As mentioned in chapter 2 and discussed in more details in chapter 5, some inconsistencies have been discovered in analysis results produced by the code parallelised by the *InSilico Research Group* [2]. Moreover, the epistasis analysis proved to be computationally expensive even for relatively small datasets, because it requires analysing of the billions of SNP combinations. It is a good candidate for the OpenMP parallelisation because it is calculating the pairwise interaction between SNPs inside the nested loop region (two loops going over all of the SNPs in the dataset, code structure is discussed in more detail in chapter 5). Additionally, the collaborating scientist (Dr. Mairead Bermingham) was also interested in incorporating this analysis in her research. Identifying the SNP by SNP interactions and accounting for them in the model would improve the model's predictive performance.

As explained in the background section epistasis is the measure of how polymorphism at one site on the genome interacts with polymorphisms at other sites. The polymorphic sites have been shown to contribute to the variation in complex traits (e.g. diseases). To date, many studies identifying the genetic basis underlying them did not account for those interactions [24]. The primary reason for that is the high computational cost associated with incorporating epistasis in GWAS [25].

This section describes the basic usage of the option, its performance on the PLINK default settings and potential use in the scientific research. The structure of the code and the parallelisation method are discussed in detail in chapter 5.

The epistasis test can be done pairwise between all of the SNPs, between the set of SNPs and all other SNPs or between two chosen sets of SNPs. In all cases the output files will only contain the results that are above a certain thresholds regulated through the additional commands. Throughout the project default PLINK threshold values have been used. To display the epistatic result, the interaction needs to be larger than or equal to 0.0001 and to count it as a significant result it needs to be larger than or equal to 0.01.

Epistasis analysis is computational expensive and so the *hapmap1* sample (83534 SNPs) and fast epistasis option have been used. The PLINK has been run with the command:

```
./plink --file hapmap1 --fast-epistasis
```

The analysis has been performed in “All x All” mode (all SNPs x all SNPs) and the total number of valid tests was 1846969599. The results were saved into two output files. The first one plink.epi.cc has six columns: chromosome of first SNP (CHR1), identifier for first SNP (SNP1), chromosome of second SNP (CHR2), identifier for second SNP (SNP2), chi-square statistics (STAT) and asymptotic p-value (P). The first four columns are self-explanatory – pair of analysed SNPs and what chromosome they are located on. The chi-square statistic is calculated as a squared standard deviation from the case that the SNPs are not affecting each other. The last column contains corresponding p-value. Extract 4 shows first six lines of this output file. The chromosome and first SNP identifier is the same for all presented entries which is

consistent with the idea of first checking all of the partners of one SNP and then proceed to the next. Because *hapmap1* sample has been picked at random, the calculated statistical measures are not representative to any population and hence we are not going to discuss them.

CHR1	SNP1	CHR2	SNP2	STAT	P
1	rs6681049	2	rs2961958	15.7	7.418e-05
1	rs6681049	6	rs4715714	19.31	1.114e-05
1	rs6681049	6	rs3805802	18.66	1.567e-05
1	rs6681049	6	rs9358001	18.66	1.567e-05
1	rs6681049	11	rs1948069	16.46	4.97e-05
1	rs6681049	11	rs361302	20.01	7.717e-06
...					

Extract 4 The first few lines of the *plink.epi.cc* file obtained from fast epistasis analysis on h*apmap1* sample.

The second output file (*plink.epi.cc.summary*) has eight columns: the chromosome of the first SNP (CHR), then SNP identifier (SNP), number of significant epistatic tests (N_SIG), number of valid tests (N_TOT), proportion of valid tests (PROP), highest statistic for this SNP (BEST_CHISQ), chromosome of best SNP (BEST_CHR) and finally the identifier of the best SNP. The proportion of valid test is simply quotient of number of significant test and number of valid tests. The best chi-square value is the highest obtained chi-square value for this SNP (the strongest interaction of this SNP). The last two columns are the chromosome on which the partner of this strongest interaction is located and its identifier. Extract 5 shows first six lines of this output file. As the file contains the summary of the epistasis analysis the strongest interaction for each SNP is recorded. If the strength of the association does not meet the requirements of the thresholds some of the columns have zeros or not a number (-*nan*) label in them.

CHR	SNP	N_SIG	N_TOT	PROP	BEST_CHISQ	BEST_CHR	BEST_SNP
1	rs6681049	847	59174	0.01431	20.01	11	rs361302
1	rs4074137	196	51513	0.003805	12.09	3	rs11716250
1	rs7540009	0	0	-nan	0	1	rs6681049
1	rs1891905	853	62736	0.0136	19.51	7	rs9638439
1	rs9729550	424	55359	0.007659	16.48	9	rs7872472
1	rs3813196	0	31865	0	4.885	3	rs13099884
...							

Extract 5 First few lines of *plink.epi.cc.summary* file obtained for fast epistasis analysis on *hapmap1* sample.

The total time taken by the fast epistasis option executed on the *hapmap1* sample with the default PLINK settings was 91 minutes and 39 seconds. There are only four contributors present in the profile shown in figure 5. Clearly, the function responsible for epistasis calculation, *Plink::calcEpistasis*, accounts for nearly all of the execution time (99.78%). As we predicted the function responsible for reading the data in takes only 0.03% (less than 2 seconds) and filtering SNPs takes only 0.01% of total time. The other functions combined together do not even take 0.2 % of the execution time.

The fact that the *calcEpistasis* function has been called only twice during the execution of the program and its overwhelming dominance suggest that parallelising this function may be very beneficial.

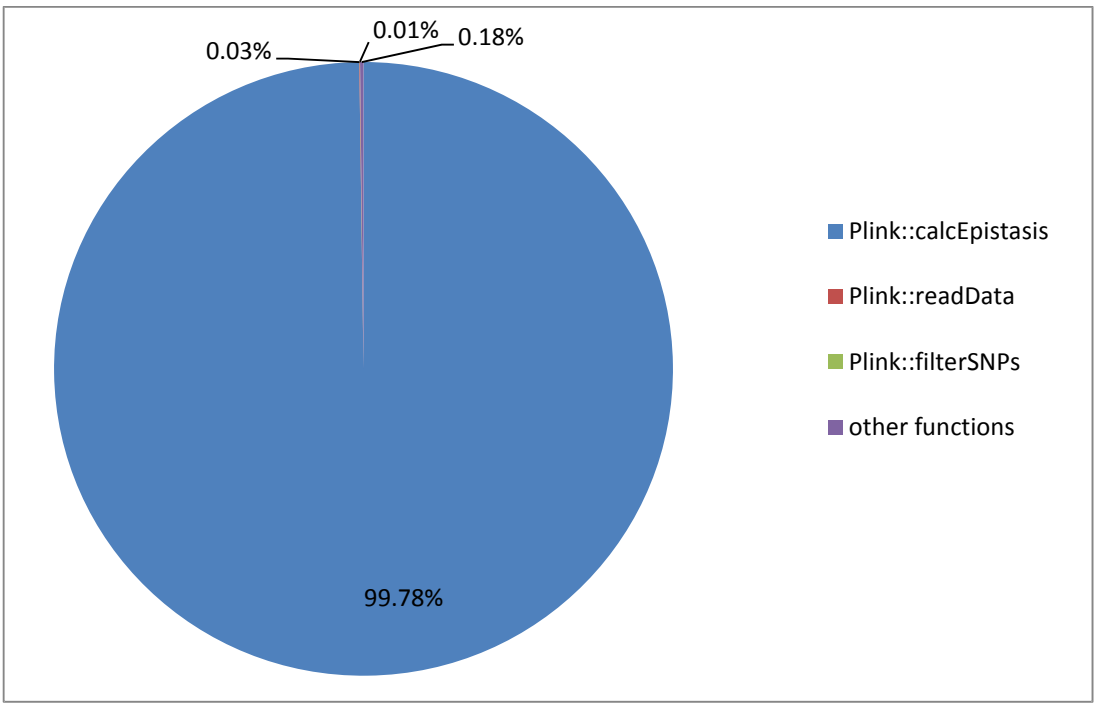

Figure 5 Profile for the fast epistasis option performed on *hapmap1* sample, with default PLINK settings and all x all mode.

We have also attempted to measure the execution time for the epistasis option for the same sample size. However, the job has been terminated after 48 hours with only about 15% portion of the sample being analysed (about 12000 out of over 83000 SNPs). Clearly, the epistasis option is much more computationally expensive than the fast epistasis, which makes it even more suitable candidate for parallelisation. The main reason behind the high computational cost is the use of linear or logistic regression, depending on whether a continuous or binary phenotype was being analysed, to calculate each pairwise association between SNPs.

PLINK's authors justify the use of fast epistasis option for computationally demanding problems because both approaches give similar results [1]. Depending on the objective of a study one may combine both approaches i.e. do the general test using the fast epistasis option and run the epistasis (using the logistic regression) on a smaller subset of SNP pairs. The two options would therefore benefit from optimisation and parallelisation, discussed in chapter 5.

3.2.2 Haplotype blocks

Due to the large number of SNPs it is more feasible and often informative to analyse simultaneously all of the markers in the region of interest. As mentioned in chapter 2, haplotype is a combination of alleles at adjacent sites on a chromosome that are inherited together. It is the strength of LD between adjacent SNPs that determine whether the two SNPs belong to the same haplotype blocks. The haplotype blocks estimation in PLINK is classified under the LD calculation options. Linkage disequilibrium is a phenomenon in which the markers display the statistical dependence. The studies suggest [26] that each chromosome can be divided into many blocks and each such block has limited number of haplotypes. Moreover, it is seems that LD is locus and population specific.

Dr Bermingham plans to use the haplotype information to reduce the redundancy in the genotype data by determining whether removing the SNPs in High LD improves the predictive performance of adopted models.

To run the haplotype blocks estimation the following command has been used:

```
./plink --tfile Genoplink_20130205 --blocks --out blocks_original
```

The same command and data sample (Genoplink_20130205) has been used by the collaborating scientist in her research. Due to the nature of the computation, only individuals with non-missing phenotype are taken into account. The default settings allow calculation of pairwise LD only between the SNPs within 200kb. This distance can be changed by addition of another command line flag.

The results of the analysis were written into two output files: *blocks_original.blocks* and *blocks_original.blocks.det.* The first one contains the list of blocks containing 2 or more SNPs. Extract 6 shows the list of SNPs making the first six blocks. Three of them consist of two SNPs only, two of them have three SNPs and one has 5 SNPs.

The second output file contains more information, stored in six columns (Extr. 7). The first one holds the chromosome identifier (CHR), the next two contain the start and end position (in base-pair units) of this block. The fourth column has the distance spanned by this block given in kilobases. The last two entries are number of SNPs in this block and their list.

```
* rs6687776 rs4970405 rs12726255
* rs11260549 rs9729550
* rs3813199 rs3766186
* rs3766178 rs3128342 rs2296716
* rs7531583 rs6681938 rs4648592 rs7525092 rs2474460
* rs12755035 rs884080
…
```

Extract 6 First few lines of the *blocks_original.blocks* file containing the haplotype blocks of the *Genoplink_20130205* sample.

```
CHR    BP1       BP2       KB      NSNPS          SNPS
  1 1030565   1049950   19.386      3          rs6687776|rs4970405|rs12726255
  1 1121794   1135242   13.449      2          rs11260549|rs9729550
  1 1158277   1162435    4.159      2          rs3813199|rs3766186
  1 1478180   1497824   19.645      3          rs3766178|rs3128342|rs2296716
  1 1706160   1844046  137.887      5          rs7531583|rs6681938|rs4648592|rs7525092|rs2474460
  1 2026361   2026749    0.389      2          rs12755035|rs884080
…
```

Extract 7 First few lines of the *blocks_original.blocks.det* file containing detailed information on the haplotype blocks of the *Genoplink_20130502* sample.

The first column for all six entries indicates that the blocks are located on chromosome 1. The starting position indicates where exactly on the genetic map they lie. The end position naturally indicates where the block ends and thus how long it is. The size of the blocks may vary significantly. Among the blocks shown in the extract 7 the shortest has 0.389 and the longest has 137.887 kilobases. They contain 2 and 5 SNPs, respectively.

The haplotype blocks analysis took 5 hours 3 minutes and 17 seconds. The log file indicates that all of the SNPs and individuals present in the sample were used in the analysis. The analysis divided 267912 SNPs into 60434 blocks.

Figure 6 shows the profile of the execution time for haplotype blocks analysis. The most dominant is *HaploWindow::enumerateGenogroups* function – it takes more than 31% of the execution time. This function is responsible for dividing the individuals present in the sample into groups of the same genotype. The second most time consuming function, taking almost 21%, is the vector class. The third and the fourth are *HaploPhase::includeIndividuals* and *HaploPhase:: phaseAllHaplotypes*

functions, respectively. The former checks whether each individual has enough non-missing genetic data to be included in the analysis and the latter performs the haplotype tests. The *two_locus_table* function (4.4%) constructs the table of independent alleles observed at two loci in all of the individuals. Next function *prunePhase* (~3%) removes unlikely regional phases from the further analysis, and *performAlternEM* function (2.22%) is used to calculate haplotype frequencies. The *other functions* label groups all of the other functions with execution time smaller than 2% of the total time. All those functions together take almost 15%. The structure of the code of above the mentioned functions is described in more details in chapter 5.

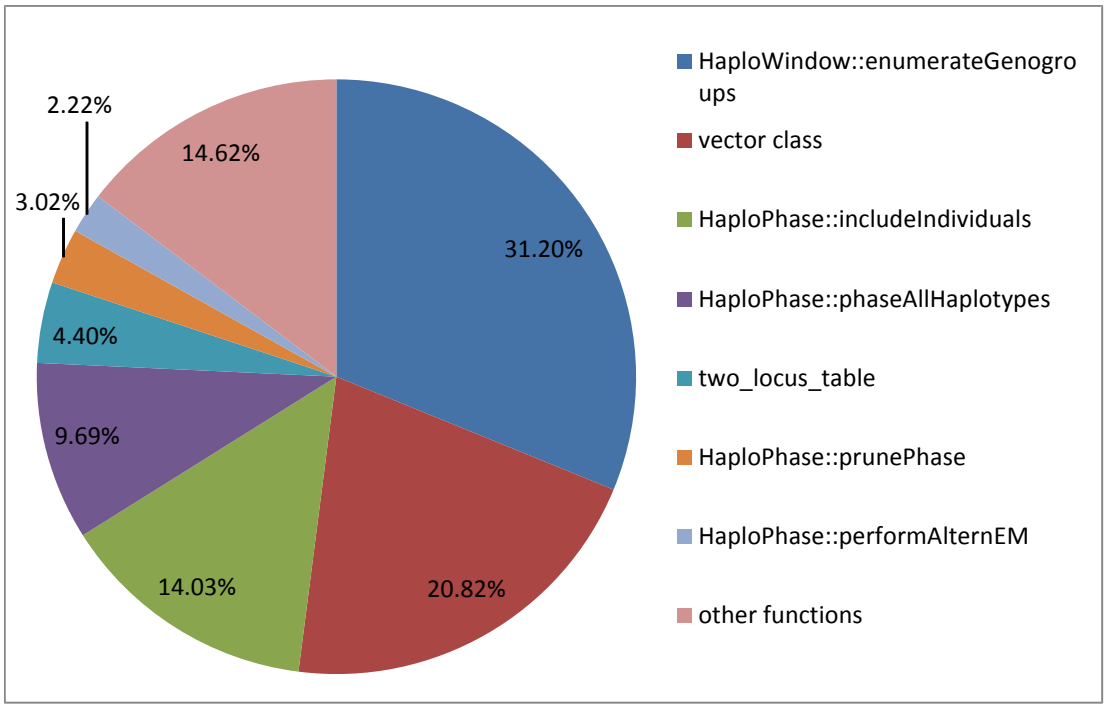

Figure 6 The profile obtained for haplotype blocks option performed on *Genoplink_20130502* sample and with default PLINK settings.

Compared with other analysed PLINK options, haplotype blocks option has clearly different structure. The main computation is not inside one function only. Moreover, there is large number of functions taking much less than 1% of the execution time. Also the contribution coming from the vector class is very noticeable. Moreover, all of the most dominant functions are called multiple times (table 1). Most of the functions were called over 6 million times and *IncludeIndividuals* and *PrunePhase* function were called almost a billion times! Consequently, the time spent in the single call for all of those functions is extremely small.

Function name	Number of calls
HaploWindow::enumerateGenogroups	6351159
HaploPhase::IncludeIndividuals	998731686
HaploPhase::phaseAllHaplotypes	6351159
two_locus_table	6351159
HaploPhase::prunePhase	998731686
HaploPhase::performAlternEM	6351159

Table 1 Number of calls to the most dominant functions of the haplotype blocks option.

Taking into account the number of functions contributing to the profile hence containing a significant part of computation, and the number of times those functions were called, it seems that the parallelisation may prove challenging. Therefore, depending on the structure of the code it may be more beneficial to concentrate on serial optimisation rather than parallelisation.

3.2.3 Conclusions

In this chapter we have presented the behaviour of four different options. The missingness rate and allele frequencies options have been executed on both artificial (*hapmap1*) and real life (*Genoplink_20130205*) data samples. The execution time of both options on small samples is dominated by the function performing the main computation of the analysis (*Plink::filterSNPs*). The contribution of other functions is also clearly noticeable (fig. 1 and 3). On the other hand, the execution times on bigger data sample are dominated by the functions reading the data into the program (fig. 2 and 4). This behaviour is believed to be typical only for the computationally inexpensive options and so the improvement of the rate of reading the data in has not become part of this project. The effect of different file formats on the execution time has also been investigated. The missingness rate analysis took the shortest time when the binary file format has been used; the transposed data format was clearly slower and normal text file was even slower. We have decided to work with the slowest file format (text format) when working on both epistasis options, fast epistasis and epistasis, and transposed format when working on the haplotype blocks option. The reason, we can use text file format for epistasis options is because of relatively small size of the *hapmap1* sample (89 individuals and 83534 SNPs). On the other hand, the size of the *Genoplink_20130205* sample is much larger (2186 individuals and 267912 SNPs) and therefore, the more efficient transposed file format has been used.

The fast epistasis and haplotype blocks options showed significantly different profiles (fig. 5 and 6) related to the different code structure. The execution time of the fast epistasis option is dominated by the *Plink::calcEpistasis* function, called twice and

taking 99.78% of the total time. This behaviour and the nature of the epistasis analysis, calculating the pairwise interactions between SNPs, suggest that parallelisation is the best approach to improving the performance. On the other hand, the haplotype blocks option seems to have the computation scattered between many different functions, which take really short time but are called enormous number of times. Depending on the code structure, the serial optimisation of the dominant functions may be the most efficient and effective way of improving the performance.

All of the initial investigations presented in this chapter has been performed with the default PLINK settings (*g++* compiler and *–O3* flag). In the next chapter the effect of different compilers and optimisation flags on the execution of epistasis and haplotype blocks options is presented.

Chapter 4

Compiler Optimisations

Before attempting any serial optimisation or parallelisation, the compiler optimisation flags have been investigated. The investigation has been conducted on hapmap1 data sample and fast epistasis option and then the results were verified using the haplotype blocks option on *Genoplink_20130502* data sample. The first part of this chapter presents the effect of different optimising flags of gnu compiler (*g++*) on fast epistasis and haplotype blocks options. The second part discusses briefly the performance of the Intel compiler (*icc*).

4.1 Gnu g++ compiler and fast epistasis option

As mentioned before, the default compiler specified in the *Makefile* is *g++* and the default flags are *-O3* and *-I*. The *-O3* flag has the highest optimisation level available in gnu compilers. However, depending on the structure of the code it not always results in the best performance. The first step of the investigation was to check how the execution time changes for the following basic optimisation flags: *-O, -O1, -O2* and *-O3*. The execution time of the fast epistasis option on *hapmap1* sample with those was respectively: 6746 s, 6400 s, 5232 s and 5499 s.

In general, *-O1* flag tries to reduce both the size of the code and the execution time. Hence none of the over 30 flags turned on by this flag take a great deal of compilation time. The *-O2* flag turns almost all of the remaining optimisation flags that do not involve a space-speed trade-off. The *-O3* flag introduces some higher level optimisations increasing both the size of the code and the compilation time. The effect of the first optimisation level is noticeable but *-O2* and *-O3* are clearly more effective. In fact, the code executes the fastest when *-O2* flag is applied. The difference between *-O3* and the *-O2* flag is ~5.5 minutes.

The next step was to determine the effect of the flags included in the third optimisation level but not in the second. These flags are: *-finline-functions, -fgcse-after-reload, -fipa-cp-clone, -ftree-vectorize, -fpredictive-commoning* and *-funswitch-loops*. Inline functions flag allows for the integration of simple functions into their callers. The *fgcse-after-reload* option eliminates the redundant load every time the reload operation is performed. This way the redundant spilling is cleaned. The *fipa-cp-clone* flag clones the function to allow stronger inter-procedural constant propagation. The *ftree-vectorize* flag performs loop vectorization on trees. Next flag, *fpredictive-commoning*, allows for the re-use of computations in previous iterations of the loops. The last flag from the third optimisation level, *funswitch-loops*, moves branches with loop invariant conditions out of the loop by creating the duplicates of the loop on both branches. PLINK has been compiled using the combination of *-O2* and those flags. Also another flag *-funroll-loops* has been investigated, it is not part of the third level optimisation but the loopy nature of the code suggests it may produce good results. This flag, as its name suggests, unrolls the loops i.e. duplicates the interior of the loop to reduce number of iterations.

Figure 7 shows execution times for all of the investigated flags and combinations of flags. Table 2 provides the description of the legend used in figure 7, as well as execution times obtained when each combination was in effect.
Considering the combination of *-O2* with single flags, the best results have been observed for *-fpredictive-commoning* (5148s), *-funswitch-loops* (5145s) and *-funroll-loops* (5175s). The fastest run obtained for *-finline-functions* flag was 5312 s, the other flags took over 5400s. Therefore, only the combinations of four most effective flags and *-O2* flag have been considered in the further investigation.

The first three combination (c1,c2 and c3) consisted of *-O2, -finline-functions* and one of the three loop flags. Only the option with the predictive commoning (c1) produced good execution time – 5167 seconds. The combinations with unswitching (c2) and unrolling loops (c3) took 5223 and 5474, respectively. The flags *-O2, -fpredictive-commoning* and *-funswitch-loops* (c4) produced the code executing in 5140 seconds. The next combination, taking 5195 second, involved *-O2, -fpredictive-commoning* and *-funroll-loops* flags (c5). The sixth combination i.e. *-O2, -funswitch-loops* and *-funroll-loops* flags (c6) was slower and took 5541 seconds. The execution time for the next three combinations (c7, c8 and c9) was slower taking: 5454s, 5549s and 5415s respectively. Combining *-O2* and three flags: *-fpredivtive-commoning, -funroll-loops, -funswitch-loops* (c10) gave the execution time of 5116 seconds. The last combination, of all five flags, produces the code executing in 5180 seconds. Clearly, the best performance is obtained when *-O2* and three loop flags are applied. The difference in the execution time obtained for the default flag *-O3* (5499s) and for the most effective combination (5116) is over 6 minutes.

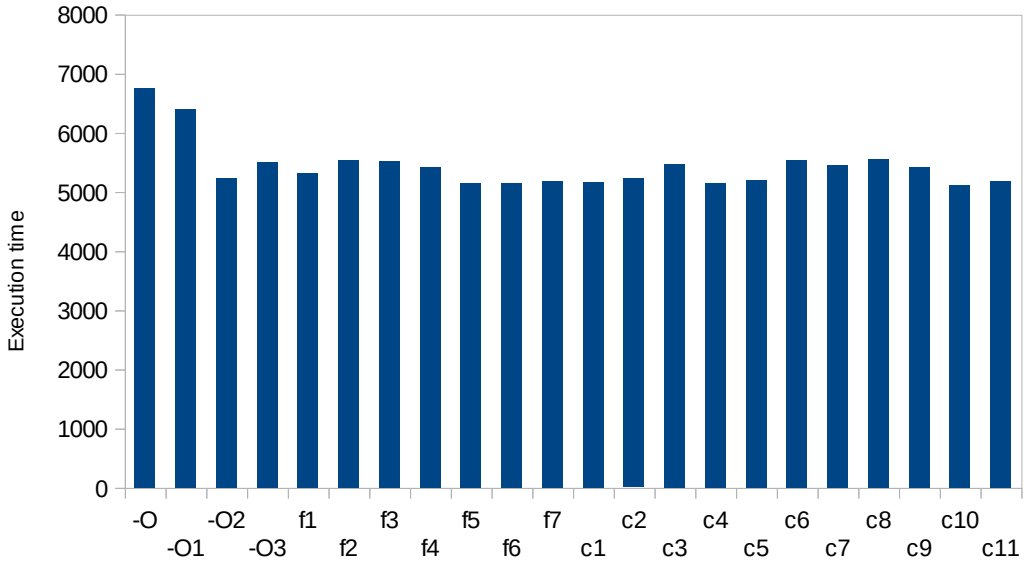

Figure 7 The execution time in seconds for fast epistasis analysis done on *hapmap1* sample when compiled with different optimisation flags. The legend is provided in table 2.

Label	Flags	Execution time (in seconds)
f1	-O2, -finline-functions	5312
f2	-O2, -fgcse-after-reload	5531
f3	-O2, -fipa-cp-clone	5517
f4	-O2, -ftree-vectorise	5409
f5	-O2, -fpredictive-commoning	5148
f6	-O2, -funswitch-loops	5145
f7	-O2, -funroll-loops	5175
c1	-O2, -finline-functions, -fpredictive-commoning,	5167
c2	-O2, -finline-functions, -funswitch-loops	5223
c3	-O2, -finline-functions, -funroll-loops	5474
c4	-O2, -fpredictive-commoning, -funswitch-loops	5140
c5	-O2, -fpredictive-commoning, -funroll-loops	5195
c6	-O2, -funswitch-loops, -funroll-loops	5541
c7	-O2, -finline-functions, -fpredictive-commoning, -funswitch-loops	5454
c8	-O2, -finline-functions, -fpredictive-commoning, -funroll-loops	5549
c9	-O2, -finline-functions, -funswitch-loops, -funroll-loops	5415
c10	-O2, -fpredictive-commoning, -funswitch-loops, -funroll-loops	**5116**
c11	-O2, -finline-functions, -fpredictive-commoning, -funswitch-loops, -funroll-loops	5180

Table 2 List of g++ compiler flags under the labels used in figure 7.
The fastest execution times are included**.**

From table 2 it can be seen that some of the flags included in the *–O3* flag have negative effect on the performance. The reason for that is the specific code structure. The flags that provide the most effective optimisations for the fast epistasis function are loop optimisations. Taking into account the structure of the function dominating the epistasis analysis, *Plink::calcEpistasis*, it makes perfect sense. However, it also implies that the haplotype blocks analysis, having completely different code structure, computation split among many small functions, may behave differently under the same set of flags and other flags might give better performance.

4.2 Gnu g++ compiler and haplotype blocks option

So far the effect of the flags has been checked only for the fast epistasis function and it is possible that the obtained results are characteristic for that function only. Therefore, the haplotype blocks analysis has been conducted when the

following flags were in effect: *-O, -O1, -O2, -O3* and combination of the flags that were identified as the most effective for the fast epistasis option: *-O2, -fpredictive-commoning, -funroll-loops* and *-funswitch-loops.* We have also decided to investigate the performance of the following combination: *-O2, -finline-functions, -fpredictive-commoning, -funroll-loops* and *-funswitch-loops.* Although, this combination proved to produce slightly slower code for the epistasis option, it might perform better for haplotype blocks option. The *-finline-functions* might be a good way to deal with small functions called enormous number of times.

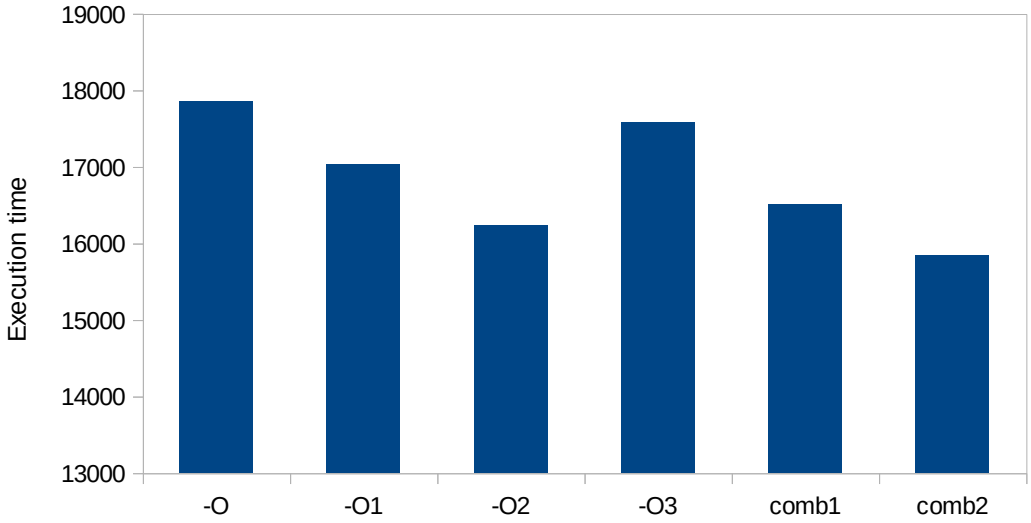

Figure 8 Effect of different compiler flags on execution of haplotype blocks analysis. The flags contained in the comb1 combinations are : *-O2, -fpredictive-commoning, -funroll-loops* and *-funswitch-loops.* Comb2 label includes: *-O2, -finline-functions, -fpredictive-commoning, -funroll-loops* and *-funswitch-loops*.

Figure 8 shows the execution times of haplotype blocks estimation with different compiler flags in effect. Each optimisation level produces coded with clearly different execution times. From figure 8, it is evident that *-O2* flag produces faster code (16234s) than the default *–O3* flag (17575s). The fastest combination for the fast epistasis option (comb1), gives the execution of 16502 seconds time, which is slower than *–O2* but faster *–O3* flags. The execution time for the second combination (with function inlining flag) produces clearly the fastest code - 15838 seconds. The reason why the first combination of flags is slower than *–O2* flag is because they are focused on the loop optimisations which are not as effective for the haplotype blocks function as they were for epistasis option. The second combination of flags is faster than *–O2* flag because it allows compiler efficient inlining of small functions. Although, *–O2* flag does not produce the fastest code for neither function, it is only slightly slower than the best combinations. Therefore, it seems to provide the optimal performance for options with different code structure. Hence, we have not investigated any other optimisation flags and decided to use *–O2* flag throughout the project.

4.3 Intel icc compiler

Lastly, the behaviour of the Intel compiler *icc* have been investigated. Again, fast epistasis option and the hapmap1 data sample have been used. The effect of the following flags has been checked: *-O0, -O1, -O2, -O3,-Os* and *-fast.* The compilation of the program failed when the *-fast* flag has been applied (error with *ipo*) in the light of the performance of other flags, the reason has not been investigated. Figure 9 shows the execution times obtained for the mentioned flags.

The first level optimisation produce the code that is 3 times faster than the code generated without optimisations (*-O0*). The performance of *-O2* and *-O3* is similar and clearly better than that of *-O1*. Although, the *-Os* flag is supposed to optimise the code for speed, its effect is less beneficial than for other level optimisations. In general, Intel compiler proved to be extremely slow. The fastest run with *-O2* flag in effect took 7568 seconds, while the same *g++* flag took 5232 seconds. Therefore, further investigation has not been attempted.

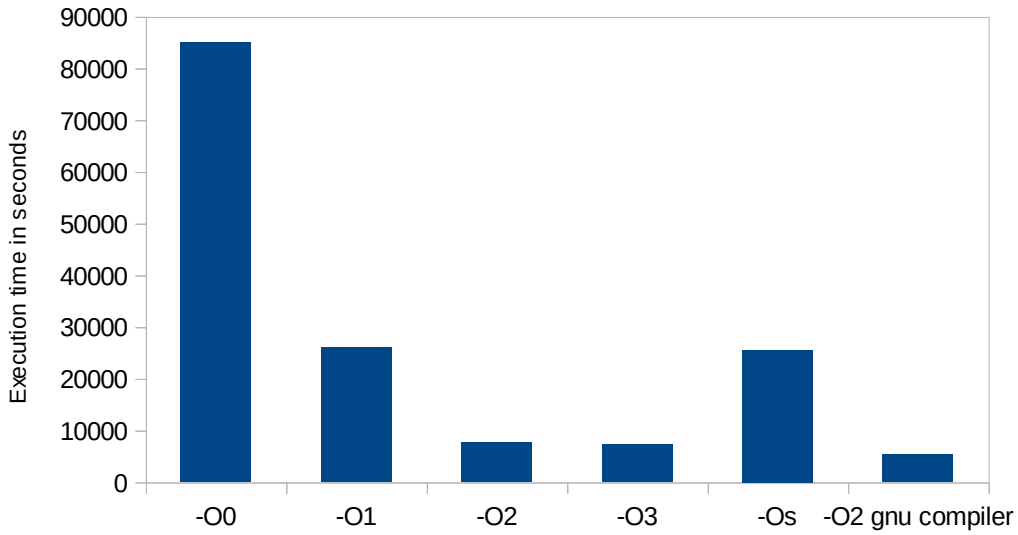

Figure 9 The effect of different *icc* compiler flags on the execution of fast epistasis option on *hapmap1* sample.

Although, the Intel compiler has been identified as less suitable for compilation of PLINK than gnu compiler, testing it still had positive consequences. Intel compiler detected several bugs in the code; all of them were of the same type and inside the same code file. One of the *icc* compiler warnings is shown in extract 8. Instead of the comparison sign '==', assign '=' has been used in the code. This misspelling occurred 12 times in the *segment.cpp* code file. Although, this file is not used by any of the PLINK options investigated during this project, the bugs have been fixed for the sake of future users and developers. Intel compiler produced several other warnings but none of them had bearings on the program execution.

```
segment.cpp(2378): warning #187: use of "=" where "==" may have been intended
  else if ( par::cnv_col = 1 )
           ^
```

Extract 8 The bug in the *segment.cpp* code file discovered by Intel compiler.

4.4 Conclusions

The flag *–O2* has been chosen as an optimal compiler configuration, even though it does not produce the fastest code for both of the investigated options. However, the code produced for both options is only slightly slower than given by their best flag configuration and still clearly faster than the default *-O3* flag. Both options have different best configurations, which is the result of significantly different code structure. Therefore, we have determined *–O2 flag* to be the most beneficial in terms of performance improvement of the analysed PLINK options. As mentioned before, the program provides a wide range of the WGAS analyses and analysing all of them is beyond the scope of this project. Therefore, it is hard to say what optimisation flags would provide the best combination for the majority of PLINK options. However, we believe the *–O2* flag is the most optimal because the usefulness of the flags it contains is not as dependent on the code structure as for higher level optimisation flags. The specific combination of flags may provide significantly better performance for a specific PLINK option, however PLINK has been designed with providing all necessary functionality required for genome-wide association studies in mind. And so it is more important to provide the optimal performance for all of its options rather than significantly better for some of them and worse for the others. Hence, *-O2* flag has been used during the course of the project.

Chapter 5

Optimising Epistasis

The structure of the *calcEpistasis* function seems to be very suitable for OpenMP parallelisation. All of the computation is taking place inside the nested loop region, which might ensure good scaling. In the first subsection we present the structure of the code and OpenMP parallelisation of both epistasis and fast epistasis options. Then the correctness of both options is discussed and finally the code scalability is presented.

5.1 Code structure

As mentioned in chapter 2, PLINK provides two ways of performing epistasis analysis. The fast epistasis option, can be applied only when the phenotype is represented as a binary trait, i.e. either affected or unaffected, and to perform it *--fast-epistasis* command needs to be used. It is an approximate test, based on a difference in association between two SNPs between cases and controls (or cases only, depending on the mode). The odds ratio measure is used to describe the strength of association between two SNPs and the value of 1.0 indicates no effect. The default analysis *--epistasis* uses either linear or logistic regression, depending on the phenotype representation (quantitative or binary). The model is based on the allele dosage for each pair of SNPs (allelic by allelic interaction). The model fits the following form: $Y \sim b0 + b1.A + b2.B + b3.AB + e$ and the strength of the interaction is based on the coefficient *b3*.

In this project, fast epistasis option has been used mainly during the parallelisation process. However, both options differ only in the way the pairwise interactions between SNPs are calculated, which as we will see, is done locally hence does not affect the parallelisation process. In fact, both options are called from within the nested loop region over pairs of SNPs, through the *if* statements (extract 9). Therefore, while describing the code structure we use fast epistasis option which profile has been discussed in chapter 3.

For both, epistasis and fast epistasis options, the function that dominates the execution time is *calcEpistasis*. It is called directly from *main* function contained in the *plink.cpp* code file. This function is responsible for performing the epistasis analysis can be divided into three functional parts. The first part sets up the output files, detects the mode of the epistatic test (AllxAll, Set1xAll, Set1xSet1 or Set1xSet2) and initializes the necessary variables. The second part contains the nested loop regions performing the epistasis analysis and produces the *plink.epi.cc* output file. The last part prints the epistatic summary statistics to the *plink.epi.cc.summary* file.

```
#pragma omp parallel for schedule(dynamic,128) private(e1, e2) reduction(+:nepi,epcc)

                for (e1=0; e1<nl_all; e1++)         // loop over all of SNPs
                 {
                  if (sA[e1])           // if SNP is in the first set
                    {
                      ...
                      for (e2=0;e2<nl_all;e2++)      // loop over all of SNPs
                      {
                        ...
                         // Skip this test under certain conditions
                     If(par::bt && par::fast_epistasis) {     //if phenotype has binary form and fast epistasis option is in effect

                        // Perform test of fast epistasis here for each individual
                        // Calculate log(OR) and SEs – Odd Ratio and Standard Error
                        ...
                         // Check this is a proper result
                        ...
                        nepi++;      // one more test performed
#pragma omp atomic
                         summary_good[e1]++;          //count as a good result for SNP1
                         If (sA[e2])
#pragma omp atomic
                           summary_good[e2]++;        //count as a good result for SNP2
                           ...
                         // Is this result the best score yet for marker in set A?
#pragma omp critical
                         {
                           if (z > best_score[e1])         // if the current z is larger than the up to date best score
                             {
                              best_score[e1] = z;          // z becomes the best score
                              best_partner[e1] = e2;           // corresponding marker becomes the best partner
                              }
                           if (sA[e2])                  // the second marker might also be in set A
                            {
                              if (z > best_score[e2])
                              {
                              best_score[e2] = z;
                              best_partner[e2] = e1;
                               }
                             }
                          }

                           // Is this worth recording?
                         If (z >= par::epi_alpha1)
                          {
#pragma omp critical
                          {
                            // printing to file plink.epi.cc
                             ...
                           }
                      }   // end fast epistasis

                    If (! par::fast_epistasis) {

                     // logistic or linear regression test for epistasis
                       …
                     }

                  }      // end of the loop over e2
                }      // end of the loop over e1
                …
```

Extract 9 Pseudo-code of section of the code in *calcEpistasis* function responsible for calculating epistatic associations between the SNPs. The bold text is OpenMP parallelisation.

The region of interest is naturally, the second part of the function – the nested loop region. The pseudo-code is shown in extract 9. The first loop (over *e1*) goes over all of the SNPs, and then, if a SNP is included in the analysis, program loops over all SNPs from the second set (loop over *e2*) to calculate pairwise associations between them. During this project all by all mode has been used i.e. both sets are the same and include all of the SNPs from the *hapmap1* sample. Also during the development and performance testing process unnecessary printing to the screen has been suppressed.

The epistatic test is skipped for the given pair of SNPs if the second SNP is not in the analysed set or if the symmetric option is in effect and the loop iteration assigned to first SNP (e1) is larger or equal than the loop iteration assigned to the second SNP (e2). The test is also skipped when certain options are enabled i.e. SNPs lie on chromosome X and if SNPs are too close (case-only epistasis). The second point makes sure that each pair of SNPs is analysed only once i.e. the results for SNP1xSNP2 are the same as for SNP2xSNP1. Naturally, the symmetric option is only enabled when both sets are the same (AllxAll and SetxSet modes).

Then if the *--fast-epistasis* option is in effect for each individual, the allelic test of a single locus is constructed. The independent alleles observed at two loci (two analysed SNPs) are counted and then the odd ratio and standard error between them is calculated. Essentially, the z value is given by the difference between odd ratios of cases and controls divided by the square root of the sum of their standard errors.

Then the number of performed test is incremented and the validity of the results is checked. If the calculated *z* value is larger than or equal to 0.01 it is counted as a significant result. Also, if the current *z* value is larger than the best score, i.e. the best up-to-date *z* value, it becomes the new best score and the corresponding SNP2 becomes best partner for SNP1. In other words, those two SNPs have the highest calculated interaction. If the second SNP is also in the first set (which is the case in All x All mode) the *z* is recorded as its best score and SNP1 as its best partner. Then, if the *z* value is greater than 0.0001 the interaction is recorded in *plink.epi.cc* file. Then the next pair of SNPs is considered.

On the other hand, if the *--epistasis* option is in effect, the logistic model is constructed if phenotype is represented in binary format. Otherwise, the linear model is applied. The strength of the interaction and other statistical variables are calculated through the calls to the member functions of the *model* object. The validity of results, best score and best partner for each SNP is calculated in the same way as for the fast epistasis option. The printing to the *plink.epi.cc* file is done slightly different and depends on the validity of the results and the phenotype representation. Then the next pair of SNPs is considered.

After all of the pairs of SNPs are analysed, the summary of results is printed to the *plink.epi.cc.summary* file.

5.2 InSilico Research Group Parallelisation

As mentioned in the previous section, the epistasis and fast epistasis analysis are invoked by the *if* statements from within the nested loop region (over two sets of SNPs). The best way to parallelise this nested loop region is through *#pragma omp parallel for* directive. That means that both options should be parallelised at the same time. Otherwise, the parallelisation of the loop region would be incomplete and might provide inconveniences for the users.

The *InSilico Research Group* [2] has only attempted the parallelisation of the epistasis option and the fast epistasis branch has not been modified. Moreover, the results of the analysis done using the parallelised code are inconsistent with the serial code results. To investigate those inconsistences, we have modified the loops so that the 5000 by 5000 SNPs are analysed, and then executed the epistasis option on both serial and parallel versions of the code. The extract 10 shows the differences between the serial and parallel versions of *plink.epi.cc.summary*. Even for this small sample, there were 18 differing lines between both files. The difference occurred in three different columns, recording: number of valid tests, proportion of significant tests and identifier of the best partner SNP. Although, the first two variables may not be that important depending on the purpose of the analysis, but the last one is crucial. The first two differences shown in extract 10 have different last columns. In the serial execution SNP rs7554746 has the strongest interaction with the rs7539462 SNP and in serial execution the best partner is rs6693272. Similarly, the SNP 6684586 has SNP rs1203650 as best partner in serial and rs1203634 is parallel versions. The last difference shown in the extract 10 is the different number of valid tests (3813 in serial and 3814 in parallel versions) and resulting difference in the proportion of the significant test.

```
2922< 1 rs7554746 0 1729 0.000000e+00 2.729000e-06 1 rs7539462
2922> 1 rs7554746 0 1729 0.000000e+00 2.729000e-06 1 rs6693272
2922: 0 0 0 0 0.000000 0.000000 0 1
…
4803< 1 rs6684586 25 3821 6.543000e-03 9.148000e+00 1 rs1203650
4803> 1 rs6684586 25 3821 6.543000e-03 9.148000e+00 1 rs1203634
4803: 0 0 0 0 0.000000 0.000000 0 1
…
4980< 1 rs593022 26 3813 6.819000e-03 9.546000e+00 1 rs740153
4980> 1 rs593022 26 3814 6.817000e-03 9.546000e+00 1 rs740153
4980: 0 0 0 1 0.000147 0.000000 0 0

there are 18 differing lines
```

Extract 10 The difference in the output files between the serial and parallelised (by InSilico Research Group) code, executed on 12 treads. Analysis has been done on 5000x5000 SNPs.

The reason behind those inconsistences is lack of a necessary reduction variables and critical sections. The modification made to the code by the InSilico Research Group are as follow: directive *#pragme omp parallel for schedule(dynamic,1) privates(e1,e2)* and one critical section have been used. The critical section is shown in extract 11.

```
                    vector_t b;
#ifdef _OPENMP
#pragma omp critical
                {
#endif
                    b = lm->getCoefs();
#ifdef _OPENMP
                }
#endif
```

Extract 11 The only critical section introduced by *InSilico Research Group* in OpenMP parallelisation of epistasis option.

Because both the *vector_t* variable and the object *lm* are declared within the parallel region, they are thread private by default. Hence, there is no need for critical section shown in extract 11. Also the *#pragma omp* directives are ignored by the compiler if the *–fopenmp* flag is not included and so there is no need to use *ifdef* conditional. The reason why the number of valid test and best partners are different for some of the interaction is because more than one thread is doing the updating at the same time i.e. classical race condition.

5.3 Parallelisation

All of the calculations are done on local variables and so there is no need to modify the code in any way to allow the OpenMP parallelisation. The *#pragma omp for schedule* has been used before the outer loop over first set of SNPs. The iterators of both outer and inner loops (e1 and e2) have been declared as private variables and two variables keeping count of performed test are declared as reduction variables.

Also two critical sections and two atomic updates have been declared. The atomic updates are used when the numbers of the significant tests for both SNPs from the pair are incremented. This way, only one thread at a time increases the count. The order in which it occurs is not important as only the final, total value of the significant interactions for each SNP is recorded in the *plink.epi.cc.summary*. The first critical section is placed around the lines of the code responsible for determining the current best *z* score and corresponding best partner. Both SNPs are considered and updated at the same time to avoid the race condition which may occur when one of the analysed

SNPs is at the same time considered by different thread with relation to other SNP. This critical section makes sure that there is only one best *z* score and best partner for each SNP at a time. The last critical section is around the printing statements. In fact, there are different modes of printing available and each is within different critical section. The critical section around the printing statements is necessary to ensure that only one thread is writing to a file at a time. Otherwise, the output would be disordered as more threads would try to write their results to the same place in file.

Although, the way of calculating the pairwise calculation in epistasis option is different (logistic and linear regression is used), the code for checking the validity of the results is exactly the same as in fast epistasis option. The printing the results to the *plink.epi.cc* is done in slightly different way but still within one critical section. There are no *#pragma omp* directives in the part of the code responsible for calculating the interactions. Therefore, all of the critical sections and atomic updates are almost in the same places in both options. In the whole *Plink::calcEpistasis* function four atomic operations and six critical sections have been used.

5.4 Testing for correctness

As mentioned before, to test the correctness of the parallelisation command *diff* and two simple tests have been used. This way the output files produced by the original (serial) version of the code have been compared with the files generated by the parallelised code. First, the *diff* command has been used and if two files have been recognised as identical no further testing has been attempted. When the *diff* command indicated that two files are different the other tests have been used. The *diff* command was useful only in cases when the order of data in both files was the same, which was only the case with *plink.epi.cc.summary* files. Those files hold the summary statistic ordered by first SNP loop iterator i.e. they are printed from within separate loop over *e1*. The order of the data in the parallel produced *plink.epi.cc* files is different because the printing is done from within loops over *e1* and *e2* (ext. 9). Therefore, depending on which thread analysed which loop iteration and how fast, the order of the data is different. However, if two *plink.epi.cc* files have the same data but recorded in different order, they are considered to be the same.

The code, we believed to be parallelised correctly, unexpectedly produced differing output files. To investigate the differences in the summary files, a simple (written in C) test has been implemented. This test reads both files in and prints the differences between them. Extract 12 shows the fragment of the difference report produced by this test for the fast epistasis option executed serially and in parallel on *hapmap1* sample. Only first and last, out of 15 differing lines are shown. The first line comes from the serially produced file, the second from execution on 12 threads and the third reports the difference in numbers. However, if the difference in the statistical values were smaller than the threshold of 0.001 they were not reported. For non-number entries like SNP identifiers, zero is assigned when they are identical and one when they differ.

```
12383< 2 rs6758984 0 14112 0.000000e+00 2.831000e+00 2 rs6714743
12383> 2 rs6758984 0 14112 0.000000e+00 2.831000e+00 2 rs10200481
12383: 0 0 0 0 0.000000 0.000000 0 1
. . .

82023< 22 rs9606603 284 62040 4.578000e-03 1.580000e+01 2 rs10933406
82023> 22 rs9606603 284 62040 4.578000e-03 1.580000e+01 2 rs778370
82023: 0 0 0 0 0.000000 0.000000 0 1
there are 15 differing lines
```

Extract 12 Few output lines of the test written to compare two *plink.epi.cc.summary* files. The first line comes from serially executed code, the second from parallel execution on 12 threads, third line reports the difference.

In all of the differing lines, only the second SNP identifier is different. This test has been repeated to observe how the reported differences change depending on number of threads. Every time the entries of first 6 columns were identical. However, the identifier of the second SNP (having the strongest epistatic interaction with SNP1) and sometimes the chromosome were different for some of the entries. As shown in extract 9 the best partner and the best *z* score are closely related to each other and determined in the same place and based on the loop iterators (*e1* and *e2*). It is also strange that all of the statistical values were unchanged regardless of the number of threads the program was executed on and only the second SNP was different. The differences have also been observed between the *plink.epi.cc* files. Because the order of the data in those files is different depending on the number of threads, the python script has been written to match the corresponding data and report the differences between them. Interestingly, all of the data from the serial file have been matched to the data from the parallel execution. However, some of the data from the parallel execution were not present in serial one. In other words, the parallel execution produced more data. The serial *plink.epi.cc* has 64586 lines containing data whereas the parallel version has 64629 lines. The parallel version has 43 more lines which suggest that when the program is executed in parallel, somehow more pair of SNPs are analysed.

To find the reason behind this behaviour, the code structure has been examined closer and special attention has been paid to the lines that could affect the identification of the second SNP and the number of analysed pairs. The symmetric option has been identified as a potential reason behind the observed behaviour. The parallel version may pick the pairs of SNPs incorrectly. To test this hypothesis, the code has been modified slightly to disable the symmetric option. Then the program has been executed serially to gain the point of reference. Before, executing the code in parallel we decided to compare the output file of the two serial versions of the code. Naturally, the non-symmetric version of the *plink.epi.cc* should contain twice as many lines as the symmetric one and their order would also be different. However, the *plink.epi.cc.summary* reports only the strongest interaction for each SNP and so it was expected to be the same for both serial versions. After all, in a non-symmetric version each pair of SNPs is analysed twice but that does not affect the final result is any way.

However, the summary files of both serial versions exhibit the same behaviour as discussed above. Again, for some of the lines the second SNP is different. Thinking about how the code execution differs for both of the serial versions, it becomes apparent than the different second SNP is a result of pairs of SNPs being analysed in different order. In parallel version the order is different because the outer loop is divided among the threads, so it is natural that for example first iteration on the eighth thread will be executed faster than 78^{th} on first thread. In the serial non-symmetric version order is different because of the way the symmetric option is implemented. When the option was in effect the test has been skipped when the first loop iterator *e1* has been larger or equal to the second loop iterator *e2* (ext. 9). In the non-symmetric version all of the pairs are analysed one by one without skipping (except *e1 = e2* case). It is clear that the symmetric option does not provide the explanation for the observed behaviour. However, testing its effect gave the indication that the order of the analysis is a crucial factor in explaining the observed behaviour. As shown in extract 9 the second SNP (the best partner) is based on the value of the *z* score. Therefore, it is possible that responsible for the strange behaviour is not the way the second SNP is determined but how the z score is calculated. If the value of the z score could be the same for different pair of SNPs (i.e. for SNP1xSNP2 and SNP1xSNP3) then the program would take the one that has been analysed first. To see if SNP1 can have the same z score for different partners, the printing statement has been introduced into the code. Now, all of the values of z that are equal to the best score are printed together with the corresponding second SNP.

```
…
SNP rs6758984 has z value of 1.68258081 and its best partner is rs10200481
SNP rs6758984 has z value of 1.68258081 and its best partner is rs6714743

…

SNP rs9606603 has z value of 3.97546195 and its best partner is rs778370
SNP rs9606603 has z value of 3.97546195 and its best partner is rs10933406
```

Extract 13 The best *z* scores for rs6758984 and rs9606603 SNPs and their corresponding partner SNPs obtained when the code has been executed on 12 threads.

Because the *z* values change many times during the execution of the program, the list of the z scores is rather long. Therefore, *grep* command has been used to find the relevant values. Extract 13 shows the highest *z* values for the two SNPs that have been shown in extract 12, rs6758984 and rs9606603. Clearly, the same *z* values have been recorded for two SNPs and the one that has been analysed first has been stored as the best partner. In original, serial version rs6758984 has rs6714743 and rs9606603 has rs10933406 as the best partners. In the parallel execution on 12 threads those pairs are analysed after the rs6758984-rs10200481 and rs9606603-rs778370 pairs and hence the observed difference.

It is unclear why the parallel version of the *plink.epi.cc* has more lines than the one produced by the serial execution of the code. We have determined it is not related to the symmetric option in effect and the way the *z score* is calculated. It is highly possible that it is an effect of the rounding error. The condition for printing a given interaction is for its z score to be bigger or equal to 0.0001, therefore depending on the order of calculation z might or might not qualify. Due to the time constraint of the project, we have not attempted to prove this statement.

The epistasis option has also been parallelised and tested. Because the epistasis option is so time consuming, the code has been modified slightly so that only 5000x5000 SNPs have been analysed. The parallelised regions are exactly the same as in the fast epistasis option, the parts of the code responsible for determining the best z score and the best partner and printing of the *plink.epi.cc* file are duplicated from fast epistasis option. Therefore we are not going to discuss them separately. The way all of the statistical values are calculated is different and so it is not possible that two pairs have the same interaction strength (i.e. *z* values are unique). The inconsistencies present in the parallelisation done by the *InSilico Research Group* have been not observed.

5.5 Scalability

Before measuring the speed gain coming from parallel execution of the code, different schedule options have been investigated on 6 and 12 threads. Running on six threads means using whole processor (six core Intel Xeon) and 12 threads is making use of whole iDataPlex server (two processors). Fast epistasis analysis on *hapmap1* sample has been executed with the following schedules in effect: s*tatic*, *dynamic*, *guided* and *auto*. Also the effect of different chunk sizes has been investigated for all of them, except *auto*.

The schedule option allows specifying which loops iterations are executed by which thread. This way all of the threads can be utilised to the fullest even if the loop has imbalanced load. The *static* schedule with specified *chunksize*, divides the iteration space into chunks (each with *chunksize* iterations) which are then assigned cyclically to each thread in order. On the other hand, *dynamic* schedule assigns the chunks of size *chunksize* on a first-come-first-serve basis. When one thread finishes processing a chunk, it receives another chunk, first on the list of unprocessed chunks. In the *guided* schedule chunks are also assigned to threads dynamically (like in *dynamic*) but they start off large and get smaller exponentially. The size of each chunk is proportional to the number of unprocessed iterations divided by the number of threads and the size of the smallest chunk is specified by the *chunksize*. The *auto* schedule leaves the assignment of iterations to threads to the runtime. If the loop is executed many times, it is possible for runtime to develop good schedule with good load balance and low overhead.

Figure 10 shows the plot of execution time against the size of the *chunksize*. The *auto* schedule has not been shown; however its performance is similar to that of guided schedule. On 12 threads *auto* schedule takes 801 seconds and on 6 threads it takes 1493 seconds. From the figure 10, it is clear that both the *static* and *dynamic* schedule have a similar performance which is not greatly affected by the size of the

chunks. There are really small differences in the execution times but the fastest execution on both 12 and 6 threads was observed for *dynamic* schedule with the *chunksize* of 128 – taking 455 s and 879 s respectively. Therefore, the *#pragma omp parallel for schedule(dynamic, 128)* directive have been used in the final version of the code (extract 9).

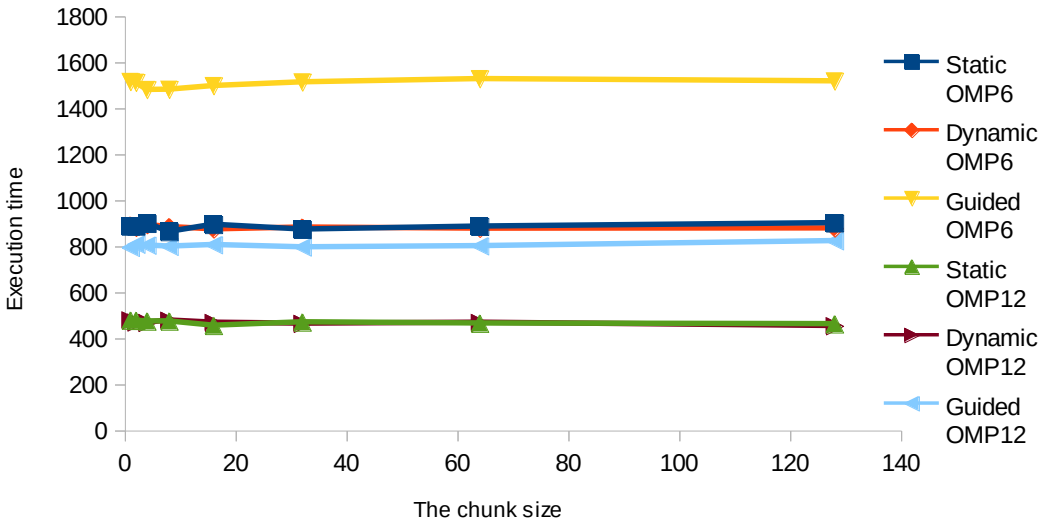

Figure 10 Effect of different schedules and chunk sizes on the execution time of fast epistasis analysis (*hapmap1 sample*) when executed on 6 and 12 threads.

The *guided* and *auto* schedules have clearly worse performance than *dynamic* and *static* with specified *chunksize*. The reason why the size of the *chunksize* has no great effect on the performance and the *guided* schedule performs so poorly is the specific structure of the code. Because the symmetric option is in effect, if the SNP1 iterator is larger or equal to the SNP2 iterator, the pair is not analysed (i.e. *e1>=e2* for iterator loops shown in extract 9). This way the iteration at the beginning of the outer loop are more expensive than the iterations towards the end where there are less SNPs to consider. Therefore, the *guided* schedule trying to lessen the burden of the last iterations performs poorly. Also, the reduction of the load of each iteration is uniform – each following iteration has one less interaction to analyse. Hence, because the number of SNPs is quite large, as long as the size of the chunk is not huge, the computation will be distributed fairly between the threads. The truly uniform distribution is only possible with the *dynamic* schedule where the chunk distribution is done dynamically. If we only take into account specific load balance, it seems that the *dynamic* schedule should perform better than *static*. The similar performance of *static* and *dynamic* schedules can be explained by the time spent on assigning the work to threads. The gain of the uniform load distribution is most likely nullified by the thread management overhead. Hence, both schedules have almost identical performance.

Figure 11 shows the speedup observed for both fast epistasis and epistasis (5000x5000) options. There is a great difference between the performance when the schedule is not specified and the best schedule. When schedule *dynamic* 128 is applied, both fast epistasis and epistasis options executed on 12 threads perform over 10 times faster than on 1 thread. That means that the original time of about 80 minutes (when flag *-O2* is applied) has been reduced to about 8 minutes. Similar results are observed for the epistasis option.

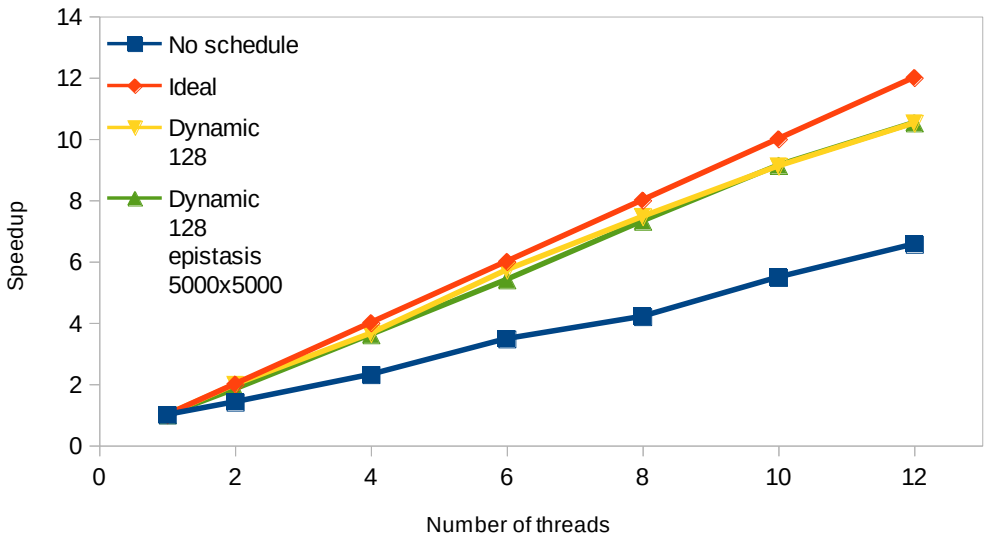

Figure 11 Speedup obtained for fast epistasis (*hapmap1*) and epistasis (first 5000x5000 SNPs from *hapmap1* sample) options.

The reason that both epistasis and fast epistasis options scale so well is because the most computationally demanding part of the calculation is inside the parallelised region. Although, there are multiple critical sections and atomic updates, they are relatively insignificant when it comes to the consumed time because they do not involve any complicated arithmetic operations. Moreover, they are not performed for all of the SNP pairs but only for those that have epistatic interaction above certain level.

To check how the code scale on different machines EPCC training machine *Morar* has been used. However, regardless of the number of threads the analysis has not been completed within the job time of 20 minutes. We have tried to execute the fast epistasis option on *hapmap1* sample on 6,8,12,16, 32 and 64 threads and all of them were aborted far from being completed (the fastest execution, on 8 threads analysed only 5 out of 22 chromosomes). It is uncertain why the fast epistasis option takes this much longer on *Morar* than on *Eddie*. Due to the time constraints of the project, further investigation in this matter has not been attempted.

5.6 Serial Optimisation

The serial optimisation of the fast epistasis option has also been attempted. Simple optimisations, like moving declaration of some of the variable outside of the nested loop region, have been tested and proved to be not very effective. The most time consuming is the part of the code responsible for building the allele dependence between two SNPs. It contains multiply nested conditional *else_if* statements with simple arithmetic operations. Re-engineering the code to remove some of the branches would be time consuming. Similarly, some of the vectors could be replaced with arrays but that would require significant modification to the code structure thus increased development time. Also because the parallelisation was so effective, any performance improvement coming from the serial optimisations would be barely noticeable. We have not investigated in details the implementation of the logistic and linear regression models and thus their optimisation have not been attempted. Due to the time constraints, they were deemed to be beyond the scope of this project. Improving the performance of those models would be very beneficial to the performance of the epistasis option.

5.7 Conclusions

Although, simple serial optimisations have been attempted, they did not provide any significant performance improvement. On the other hand, parallelisation proved to be very effective. The correctness of the parallelisation has been confirmed through extensive testing. The consistencies observed in the results for the *InSilico Research Group* have not been observed in our parallelisation. The fast epistasis option produces slightly different results when executed on different number of threads. These inconsistences have been proved to be the results of the algorithm responsible for calculating the strength of the interaction. In the implemented calculation it is possible for a given SNP to have the same strength of interaction with more than one SNP, and the best partner becomes the one that have been analysed first.

The serial version of the fast epistasis code executes in about 90 minutes while the parallelised code on 1 thread executes in about 80 minutes. The parallelised code executed on a single thread has a significantly shorter execution time. This implies that the *–fopenmp* flag introduces some changes in the code that allows compiler to optimise the code more effectively. On 12 threads, the code is executed in less than 8 minutes, which is over 10 times faster than when executed on 1 thread. The epistasis option scales equally well. The execution time of epistasis analysis including 5000x5000 SNPs on 1 thread is 1800s and on 12 threads it is 171 seconds. That means that the execution on 12 threads is 10.5 times faster than on 1 thread.

Both options have benefited greatly from the OpenMP parallelisation – running them on 12 threads reduces the execution time over 10 times. Because the parallelisation has been so effective and simple while serial optimisations proved to be ineffective, further optimisations have not been attempted.

Chapter 6

Optimising Haplotype blocks

In this chapter we will first describe the structure of the code and the functions call tree. Due to the structure of the code, OpenMP parallelisation is not as beneficial in the epistasis option case. Therefore, the main focus is on serial optimisations. The modifications to the most time consuming functions are presented in separate subsections. Due to the large number of optimisations tested, only the most significant ones are discussed. Secondly, the parallelisation attempt is described briefly and then the overall improvement in performance is presented.

6.1 Code structure

As mentioned in chapter 3, the structure of the code of the haplotype blocks option is very different from the epistasis option. The *main* function calls *Plink::mkBlks* function which contains the loops over the chromosomes and regions of DNA considered in kilobases. This function uses *LDPair* class and *PairwiseLinkage* to calculate LD and confidence interval (CI) between the SNP pairs and then makes a list of strong LD pairs within the analysed region. Next, the haplotype blocks are constructed.. The blocks cannot overlap, need to have at least 2 SNPs, cannot be too long in bases compared to their size in markers (if they are long in bases they need to have many markers) and 95% of informative markers need to have strong LD. If the block meets these criteria, it is added to the block list in order by first marker number. Finally, the blocks and their information are printed into output files (**.blocks* and **.blocks.det*)

The most time consuming part of *mkBlks* function is LD calculation invoked by calculateLD function which contains a single call to another function – *dprime* shown in the extract 14.

```
void PairwiseLinkage::calculateLD()
{
   dp = PP->haplo->dprime(a,b);
}
```

Extract 14 *CalculateLD* function, called from within the *mkBlks* function.

```
double HaploPhase::dprime(int l1, int l2)
{
  calculateDp = true;
  double dp = rsq(l1, l2);
  calculateDp = false;
  return fabs(dp)
}
```

Extract 15 *Dprime* function called from within the *calculateLD* function.

The function *HaploPhase::dprime*, shown in extract 15, calls *HaploPhase:: rsq* function which then calls *HaploPhase::phaseAllHaplotypes*. The last function is present in the haplotype blocks profile shown in figure 6. All the other functions having considerable contribution to the total execution time are called from within the *phaseAllHaplotypes* function.

6.2 Optimising dominant functions

There are six functions named in the haplotype blocks execution profile shown in figure 6. They are: *HaploWindow::enumerateGenogroups* (31.2%), *HaploPhase:: includeIndividuals* (14.03%), *HaploPhase::phaseAllHaplotypes* (9.69%), *two_locus_table* (4.4%), *HaploPhase::prunePhase* (3.02%) and *HaploPhase:: performAlternEM* (2.22%). The optimisation of all of them has been attempted, with the most time spent on the most dominant functions. In the following subsections code structure and attempted optimisations are discussed.

6.2.1 *EnumerateGenogroups* function

This function is located in a separate code file called *genogroup.cpp*. It is responsible for collapsing all genotypes into unique groups - *genoGroups*. This way the subsequent analysis is performed on these entities rather than on individuals. Extract 16 shows the original code. It loops over all of the individuals and groups them with respect to their genotype. Only the representative individuals (founders) with non-missing genetic information are considered. Then the genotype set is built using *MultiLocusGenotype* class. It consists of bool vectors *g* and *skip*, and integers *count* and *reference*. Vector *g* holds the allele codes, the *count* holds the number of individuals with the given genotype, and *reference* stores the identifier for each group. Vector *skip* is not used in this function.

```
void HaploWindow::enumerateGenogroups()
{

 for (int i=0; i < P->n ; i++)             //consider each individual
 {
    if ( ! (P->sample[i]->founder && haplo->include[i]))          //only phase non-missing founders
      continue;

   MultiLocusGenotype * m = new MultiLocusGenotype;      // build new multilocus genotype set

   if ( haplo->X )
     m->g.push_back(P->sample[i]->sex);                  // include sex if X chromosome is analysed


   for (int s = 0; s < ns; s++)                      // analyse genotypes
   {
     bool s1 = par::SNP_major ?
     P->SNP[ S[s] ]->one[i] :
     P->sample[i]->one[ S[s] ];

     bool s2 = par::SNP_major ?
     P->SNP[ S[s] ]->two[i] :
     P->sample[i]->two[ S[s] ];

     m->g.push_back(s1);
     m->g.push_back(s2);
   }


  m->count = 1;
  m->reference = i;

// But have we already seen a similar genoGroup?
  set<MultiLocusGenotype*>::iterator im = genotypes.find(m);
  if (im == genotypes.end() )
   {
     genoGroup[i] = m;
     genotypes.insert( m );
   }
   else
   {
    delete m;
    (*im)->count++;
    genoGroup[i] = *im;
   }


  } // Next individual
}
```

Extract 16 Unmodified *genogroup.cpp* code file.

If the chromosome X is taken into consideration, then vector *g* holds additional element. The SNPs are coded depending on their allele frequency and pushed back onto the vector *g*. The object *intvec_t S* holds the list of SNP numbers and the *vector<individual*> sample* stores the genotype information for each individual.

Next, the *count* is set to 1 and *reference* to *i* (number assigned to the current individual). In other words, at this point it is assumed that this individual is the first one belonging to this group. Then all of the previously analysed groups are searched to determine whether the current group is unique or has been recorded before. If the group is unique then the current *m* is assigned to the *genoGroup i* and this new group is

inserted into *genotypes* set. The *genoGroup* is a vector of type *MultiLocusGenotype** and clearly holds the information which *genoGroup* each individual belongs to. The *set<MultiLocusGenotype*> genotypes* stores information about each unique group (i.e. its reference number, number of individuals belonging to it and genotype data). If the similar *genoGroup* already exists then the currently processed individual is assigned to it and the current *m* is destroyed. Then the next person is analysed.

The first modifications are shown in extract 17. First the bool variables *s1* and *s2* were declared outside the loop over *s*. Then the ternary conditional operators were replaced with *if...else* statements. Now both Boolean variables are declared only once for each individual and the conditional statement is calculated only once for each *s*. These changes did not improve the performance greatly because in the used data set (*Genoplink_20130205*) the variable *ns* (number of SNPs in haplotype) is equal to 2. The samples with larger values of *ns* would benefit more from this modification.

Also, to avoid referencing global *P->n* (number of individuals in the sample) with each iteration of the loop over individuals, it has been stored in the local variable *size* (extract 17).

```
...
int size = P->n;
bool s1,s2;
for (int I = 0; i < size; i++)
  {
    ...
    for(int s=0; s<ns; s++){

      if (par::SNP_major){

        s1=P->SNP[S[s]]->one[i];
        s2=P->SNP[S[s]]->two[i];
        m->g.push_back(s1);
        m->g.push_back(s2);
      }
      else{
        s1= P->sample[i]->one[ S[s] ];
        s2= P->sample[i]->two[ S[s] ];
        m->g.push_back(s1);
        m->g.push_back(s2);
      }
  }
  ...
}
...
```

Extract 17 Modifications made to the *genogroup.cpp* code file.

Push backs are more expensive than direct access to the memory locations. Therefore, we have attempted to replace the push backs with direct writes. Vector g has been resized to have *2*ns* (or *2*ns + 1* if *haplo->X* is true i.e. X chromosome is also being analysed) elements. Then both *s1* and *s2* were written directly into the corresponding elements of *g*. However, this modification significantly increased the execution time. This is because resizing of the vector was much more time consuming than the push backs. The time spent in each part of the function has been measured and printed to screen. The resizing of the vector *g* took 0.00000852 seconds and the time

spend in the loop over *s* was 0.00000036 seconds. The time spend inside the loop over *s* in the original code was 0.00000048 seconds. Clearly, the time gained by replacing the push backs with the direct memory accesses does not compensate the time spend on resizing the vector.

The reason for that is relatively small number of push backs (*ns = 2* only). Also because the vector can be resized only after it has been declared, the resizing has been done for all of the individuals i.e. inside the loop over *i*. Therefore to reduce the resizing time, the resizing would have to be done only once for all of the individuals i.e. outside the loop over *i*. Consequently, the declaration of *m* should be placed before the loop as well. The problem is *m* is dynamically allocated and so all of the memory accesses to it are done through pointers. Moreover, *genoGroup* is a vector of pointers to *m* and it is used not only by *enumerateGenogroups* function but by other unrelated functions as well. Thus all of the relevant *m'*s need to exist at the same time. It is not possible to declare only one *m* and reuse it for all individuals without significant re-engineering of the code in several different functions. Due to the time constraints further investigation into this optimisation has not been conducted.

The last modification attempted was moving the condition *haplo->X* outside the loop over individuals (over *i*). It is possible because the condition is not dependent on any variable inside the loop. This modification requires duplication of the code but results in the condition being calculated only once. However, this modification slowed down the execution considerably.

```
. . .
if ( haplo->X )                //if chromosome X is being considered
  {
    for (int i=0; i < P->n ; i++)      //consider each individual
     {
       ...
       m->g.push_back(P->sample[i]->sex); // add sex
       …
     }
else {
     for (int i=0; i < P->n ; i++) {         // consider each individual, do not analyse chromosome X
       ...
     }
```

Extract 18 Modification done to the *enumerateGenogroups* function – loop independent condition has been moved outside of the loop.

It is possible that because the *HaploPhase::includeIndividuals* is rather small function, the compiler was doing a rather good job inlining it. However, increasing the size of the function (duplication) made it less effective. Therefore, only the modifications shown in extract 17 have been used in further development and performance analysis. The code with those changes has been executed in 13421 seconds which is 2813 s (46 min) faster than the unmodified code compiled with –O2 flag (16234).

6.2.2 *IncludeIndividuals* function

The *HaploPhase::includeIndividuals* function takes over 14% of the total execution time. It is responsible for checking the genotype information for a particular individual and if certain conditions are met or the person has too much genotype data missing they are excluded from the subsequent analysis. The most significant part of the code is shown in extract 19. Only two changes have been made in this function. Boolean vectors *s1* and *s2* have been replaced with Boolean scalars. Then as a consequence of the first modification, two loops over *s* have been merged. The first step required the second because the second loop re-uses the results of the first loop. Replacing vectors with scalars is possible because the allele codes are used only to calculate the amount of missing genotype data (*nm*). Therefore they are needed only locally, and merging both loops ensures the required locality.

```
void HaploPhase::includeIndividuals(int i)
{
// Do not look at non-reference individuals in some circumstances
…
vector<bool> s1(ns);
vector<bool> s2(ns);

// Flipping allele-coding for homozygotes
for (int s=0; s<ns; s++)
  {
   if (par::SNP_major)
    {
     s1[s] = P.SNP[S[s]]->one[i];
     s2[s] = P.SNP[S[s]]->two[i];
    }
  else
   {
    s1[s] = P.sample[i]->one[S[s]];
    s2[s] = P.sample[i]->two[S[s]];
   }
   if (s1[s] == s2[s])
   {
     s1[s] = !s1[s];
    s2[s] = !s2[s];
   }
  }

 // Count amount of missing genotype data at this position
int nm = 0;
for (int s=0; s<ns; s++)
    if (s1[s] && !s2[s])
      nm++;

// if too much genotype data is missing do not include this individual
...
}
```

Extract 19 Unmodified *HaploPhase::includeIndividuals* function.

The individual effects of those changes are not big however, because the *includeIndiviudals* function has been called almost a billion times (table 2) they accumulate and have a positive effect on the overall performance. The modified code (extract 19) executes in 12831 seconds which is 3403 s (over 56 min) faster than the unmodified code and almost 10 min faster than the previous version. Also replacing vectors with scalars reduced contribution from the vector class.

6.2.3 *PhaseAllHaplotypes* function

This function took almost 10% of the execution time and is responsible for the main part of the estimation. All the other functions of the *HaploPhase* type are called from within this function. The function calculates the haplotype frequencies. The first step is to define the regions of the analysis and the SNPs they contain, then the haplotypes (*HaploPhase::enumerateHaplotypes*), genoGroups (*HaploPhase:: enumerate Genogroup*s) and phases (*HaploPhase::enumeratePhase*) are enumerated. Next the EM algorithm (Expectation-Maximisation) is used to estimate the haplotype frequencies (*HaploPhase::performAlternEM*), after which the unlikely regional phases are pruned (*HaploPhase::prunePhase*). The last part involves reporting haplotype frequencies and haplotype phase probabilities, and performing the haplotype association tests (*HaploPhase::performHaplotypeTests*).

Quite a large number of small modifications have been tested. However, it was hard to determine their effect on the overall performance. Not only, the changes were small but also most of the optimisation work has been tested on the subset of two chromosomes which made it even harder to determine their effect. Therefore, taking into consideration the time necessary to test all of the modification on the whole sample (3 measurements required) and the fact that the function takes only 10% of the execution time, we have decided to test the changes that had chance to clearly improve the performance.

In the end, after development work on the fragment of the data set (2 chromosomes only), the only adopted changes were: common sub-expression elimination (instead of *P.n* – number of individuals, integer *Indiv_num*), moving some of the variable declarations outside of the loops and replacing the power function call with multiply sign. Therefore, we have reduced the number of declarations, improved locality by using local variable instead of global and reduced the number of small function calls. Observed for the entire sample timings were very inconsistent and the best one was only 3 minutes faster (12659) compared to the previous code version with unmodified *phaseAllHaplotype* function (12831). The slowest was almost 15 minutes slower. Thus we have decided to leave the *phaseAllHaplotype* function unmodified.

6.2.4 *Two_locus_table* function

The *two_locus_table* function is not part of *HaploPhase* class. It is defined inside the *helper.cpp* code file and it is being called from *mkBlks* function. This function is used only by the haplotype blocks option. It counts the independent alleles observed at two analysed loci for all of the individuals present in the sample. Because *two_locus_table* function takes only 4.4% of the total execution time only the optimisations that could clearly improve the performance at the small development cost, have been considered. Three changes have been made. Extract 20 shows the relevant part of the code before and extract 21 shows the same part of the code after modifications. First of all, the declarations of the Boolean scalars have been placed outside of the loop over individuals. Secondly, the four ternary conditional statements have been replaced by one *if…else* conditional statement. Lastly, local integer variable *size* has been declared to hold the number of individuals (*PP->n*).

```
 …
for (int i=0; i< PP->n; i++)
{
   Individual * person = PP->sample[i];
   If ( person->missing || ! person->founder )
     continue;

   bool a1 = par::SNP_major ? PP->SNP[l1]->one[i] : person->one[l1];
   bool a2 = par::SNP_major ? PP->SNP[l1]->two[i] : person->two[l1];
   bool b1 = par::SNP_major ? PP->SNP[l2]->one[i] : person->one[l2];
   bool b2 = par::SNP_major ? PP->SNP[l2]->two[i] : person->two[l2];
   …
```

Extract 20 Part of the original code of *two_table_locus* function located in *phase.cpp* code file.

```
   bool a1, a2;
   bool b1, b2;
   int size = PP->n;

   for (int i=0; i<size; i++)
      {
       Individual * person = PP->sample[i];
       if ( person->missing || ! person->founder )
         continue;

      if(par::SNP_major){

         a1 = PP->SNP[l1]->one[i];
         a2 = PP->SNP[l1]->two[i];
         b1 = PP->SNP[l2]->one[i];
         b2 = PP->SNP[l2]->two[i];
      }
     else{
         a1 = person->one[l1];
         a2 = person->two[l1];
         b1 = person->one[l2];
         b2 = person->two[l2];
     }
   …
```

Extract 21 Part of the modified code of *two_table_locus* function located in the *phase.cpp* code file**.**

The above described modifications have improved the performance of the haplotype blocks option. The code executed in 12367 s which is 464 seconds (almost 8 min) faster than the previous version of the code – 12831 (with modified *enumerateGenogroups* and *includeIndividuals* functions).

6.2.5 *PrunePhase* and *performAlternEM* functions

Because both of those functions are not very time consuming (*prunePhase* – 3.02% and *performAlternEM* – 2.22%) we did not spend much time on optimising them. The *performAlternEM* function contains the Expectation-Maximisation algorithm and thus has rather complicated code structure. Taking into consideration the time constraints of this project, only simple modifications have been tested. They were: the common expression elimination for the number of individuals present in the analysed sample (accessed through *P.n*) and moving time consuming declarations (vectors) out of the loops and nested regions.

In *prunePhase* we attempted to replace push backs to multiple vectors with direct memory accesses. However, some of the variables modified inside the loop had an impact on the vector sizes and since the resizing of vectors before the loop and replacing them with arrays became more complicated, no significant modifications have been made to this function.

Those changes resulted in code executing in 12849 seconds which is slower than the code after the last adopted modification (12367s). Therefore, both *prunePhase* and *performAlternEM* functions have not been modified in the final code version.

6.3 Serial Optimisation results

The overall performance improvement after each major optimisation stage is shown in the figure 12. Changing the optimisation flag from *–O3* to *–O2* reduced the execution time from 17575 to 16234 seconds and introducing the changes into the *enumerateGenogroups* functions (discussed in section 6.2.1) resulted in the execution time of 13421 seconds. The changes made to the *includeIndividuals* and *two_locus_table* functions gave the code executing in 12831 and 12367 seconds, respectively.

The initial execution time of 4 hours and 53 minutes has been reduced to 3 hours and 26 minutes. Thus the performance of the haplotype block options has been improved by about 30%.

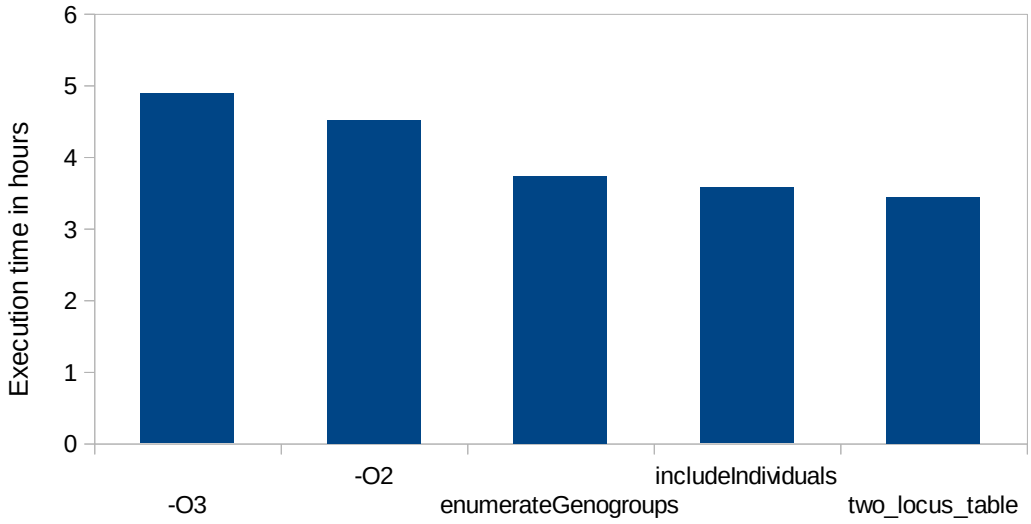

Figure 12 Execution times measured after each major optimisation stage.

6.4 Parallelisation

Unlike in the epistasis option, the haplotype blocks option does not have any obvious regions for parallelisation. Parallelising the functions present in the profile (fig. 6) would not improve performance because the time needed to spawn the threads is larger than the time needed to execute those functions. Therefore, the only region potentially suitable for OpenMP parallelisation is within the *mkBlks* function, shown in extract 22. The first loop (over the chromosomes) is not the best candidate for using the *parallel for* directive because there are only 23 chromosomes in human genome. Hence the loop is not big enough to benefit significantly from parallelisation. The loops over the analysed regions (200kb – default settings) and the intervals (depending on the distance between the SNPs within the region) are much bigger and so may be parallelised. Extract 22 shows the introduced parallelisation. The *parallel for* directive has been used and two variables *x* and *y* (loop iterators) have been declared as private. Three critical regions have been defined. The first one encompasses the calculation of linkage disequilibrium and confidence interval for each analysed pair of SNPs. Both functions *calculateLD* and *caclulateCI* contain the calls to the functions using global variables and so they need to be executed by one thread at a time. The other two functions involve inserting objects into map and set. The *dpStore* is a map of pairs and their corresponding *LD* and *CI* coefficients and *StrongPair* is a set of pairs of SNPs with strong *LD* and distance between them. To eliminate the possibility of two threads trying to perform writes to the same space in the memory (conflicting stores) both inserts have been placed within the critical sections. Figure 13 shows the resulting speedup. On 2 threads the execution time is 11185 seconds and on 12 threads it is

10901 seconds. The execution time on 2 threads is almost 20 minutes faster than on 1 but adding more threads does not reduce the execution time significantly. There is almost no speedup.

```
Map<Range, vector<int> > Plink::mkBlks(int null1, int null2)
  {
   ...
  for(int chr = startChromosome; chr <= finalChromosome; chr ++)          // loop over all chromosomes
   {
     ...
#pragma omp parallel for private(x,y)
      for(x = fromPosition; x < toPosition; x++)                  // loop over all regions
        {
         ...
         for(y = x+1; y <= toPosition; y++)                       // loop over smaller segments within each region
           {
            ...
#pragma omp critical
                        {
                         PairwiseLinkage thisPair(x,y);
                         thisPair.calculateLD( );                    // calculate linkage disequilibrium coefficient for pair x,y
                         thisPair.calculateCI( );                    // calculate the confidence interval for pair x,y
                         }
              ...
#pragma omp critical
                        {
                         dpStore.insert(make_pair(t,d));        // t is x,y pair and d consists of corresponding coefficients
                        }
             ...
#pragma omp critical
                        {
                         strongPairs.insert(p);                  // p stores pair x,y and the distance they span
                         }
            ...
```

Extract 22 Code of the *mkBlks* function. The bold text shows introduced parallelisation.

The lack of time prevented us from investigating the effect of different schedules. However, the code structure suggests that even the most suitable schedule would not improve the performance significantly. The only way to improve the performance of the parallelised code is to reduce the size of the critical section encompassing the LD calculation. That would require moving the critical section deeper into the functions call tree. As discussed at the beginning of this chapter, *calculateLD* function calls *dprime* function, which calls *rsq* function, which calls *HaploPhase:: phaseAllHaplotypes* function. The last function calls all the other functions present in the profile (fig. 6). To locate all the regions that could cause the race condition when not executed inside the critical section, the Valgrind's DRD tool (thread error detector) [23] has been used. After reporting 10000000 detected potentially conflicting load and stores, the program stopped reporting. Because of the time constraints of the project we were unable to investigate detected errors. Therefore, reducing the size of the critical section is deferred to the future development works.

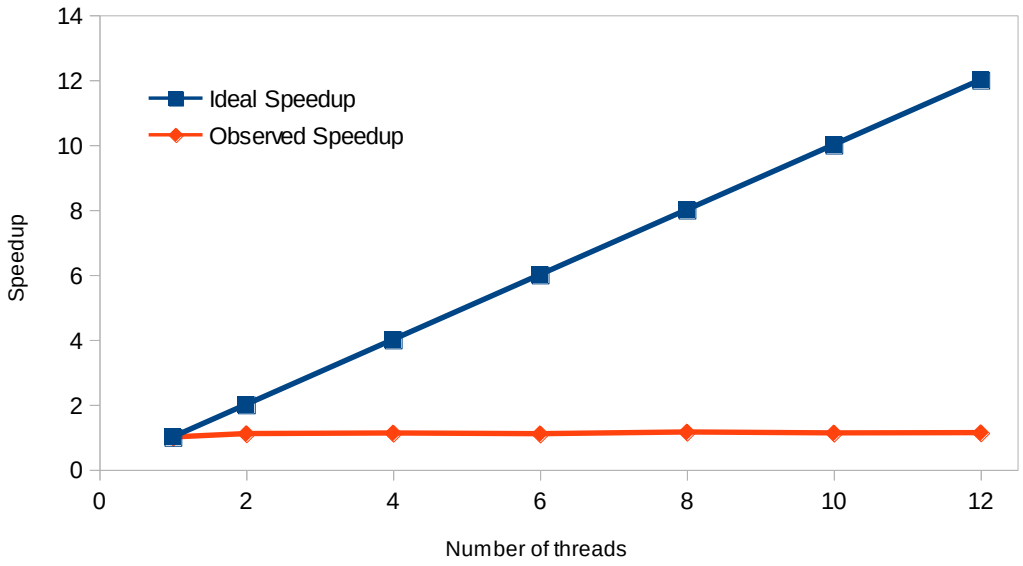

Figure 13 The speedup obtained for the haplotype blocks option.

6.5 Conclusions

During the optimisation process, only the modifications clearly improving the performance have been adopted. Three functions have been optimised: *enumerateGenogroups*, *includeIndividuals* and *two_locus_table*. Although the optimisation of other functions has been attempted, it did not result in performance improvement. The final version of the code has been executed in 12367 seconds which is almost 1.5 hour faster than the original code (17575s).

The parallelisation of the haplotype blocks option proved to be challenging. The only region suitable for parallelisation is inside the *mkBlks* function. The parallelised code scales very poorly because the most computationally expensive calculations are performed within the critical section. Due to the time constraints, the investigation into the effect of different OpenMP schedules and possibility of reducing the size of the critical section by moving it deeper into the function call tree has not been attempted.

Chapter 7

Conclusions

In this project we have worked on improving performance of two PLINK options – epistasis and haplotype blocks. First, to understand how the PLINK works and what functionality it provides, PLINK tutorial has been analysed. Some of the tutorial cases have been executed on artificially small data sets and profiled. Then both epistasis and haplotype blocks options have been profiled as well. The haplotype blocks option has been always performed using the real life data set consisting of 267912 SNPs from 2186 individuals (*Genoplink_20130205*) and the epistasis has been always performed using the artificially smaller data set (83534 SNPs from 89 individuals – *hapmap1*).

The first step of the optimisation was investigation of different compilers and optimising compiler flags. The behaviour of the default Gnu *g++* and Intel *icc* compilers and their different flags have been studied. It has been determined that the most optimal performance for both analyses has been given by the *g++* compiler and *–O2* optimisation flag. Although both options had the combinations of flags that produced slightly faster code than *–O2* flag, they were different for each option. Therefore, in order to provide good performance for all of the PLINK options the second optimisation level has been adopted.

The profiles of epistasis and haplotype blocks options differ significantly. Those differences are the result of different code structure. To gain the best improvement in the performance different approaches have been adopted for both options.

The epistasis option has been successfully parallelised and proved to scale very well. Both methods of performing the epistasis analysis showed the speedup of about 10.5 when executed on 12 threads and using schedule dynamic with the *chunksize* of 128. The correctness of parallelisation has been tested and the explanation for all of the inconsistences has been given. The simple serial optimisation of the fast epistasis analysis did not produce clear improvement in the performance, and thus further investigation has not been attempted. The performance of the normal epistasis (using the linear or logistic regression) could be improved by optimising the implementation of the regression models.

Due to the specific code structure of the haplotype block option, the focus has been put on serial optimisation. The modifications that proved to be the most beneficial were: replacing the ternary conditions with the *if…else* statements, replacing the Boolean vectors with Boolean scalars and removing some of the declarations from inside the loops. The initial time of about 5 hours has been reduced to about 3.5 hours, which means the performance has been improved by 30%.

There was no obvious region for parallelisation inside the code of haplotype blocks options. It has been parallelised but because of the specific code structure it did not scale well. Even though the functions present in the profile have many loops and nested loop regions, parallelising them would not be beneficial because the time spent within those functions is comparable to the time required to spawn the threads. The

#pragma omp parallel for directive has been used inside the *mkBlks* function but because the most computationally expensive part of the calculation has to be inside the critical region, the performance improvement is hardly noticeable regardless of the number of threads used.

Any future attempts on the parallelisation of the haplotype blocks options should focus on reducing the size of the critical section by moving it deeper inside the function calls. The performance of this analysis would also benefit greatly from reducing the number of vectors and vector operations used in the code.

Appendix A

Example job script

The contents of the example job script *haplo.sh*:

```
#!/bin/sh
#$ -l h_rt=06:00:00
#$ -cwd
#$ -M mantraani@yahoo.pl
#$ -m abe
#$ -N haplo_OMP1
#$ -pe OpenMP 1


. /etc/profile.d/modules.sh
export OMP_NUM_THREADS=$NSLOTS
echo OMP_NUM_THREADS = $NSLOTS

make clean
make

./plink --tfile /exports/work/physics_epcc_msc/s0789793/Genoplink_20130205 --blocks --out blocks_mod1
```

References

[1] S. Purcell at al. 2007. *PLINK: A tool set for whole-genome association and population-based linkage analysis* Am. J. Hum. Genet. 81, 559-575. Online at: http://pngu.mgh.harvard.edu/~purcell/plink/

[2] *InSilico Research Group.* Online at: http://insilico.utulsa.edu/ and https://github.com/insilico/plink (referenced on 29/03/2013).

[3] *Human genome project.* Human Genome Project Information Archive. Online at: http://web.ornl.gov/sci/techresources/Human_Genome/index.shtml (referenced 20/08/2013).

[4] *International Hapmap project.* Online at: http://hapmap.ncbi.nlm.nih.gov/ (referenced 21/08/2013).

[5] L. Kryglyak, D.A. Nickerson, 2001. *Variation is the spice of life*. Nat. Genet. 27, 234-235.

[6] J.C. Venter et al. 2001. *The sequence of the human genome*. Sci. Sign. 291, 1304.

[7] *Whole-genome association studies.* National Human Genome Research Institute. Online at: http://www.genome.gov/17516714 (referenced 17/07/2013).

[8] A. Galvan et al. 2010. *Beyond genome-wide association studies: genetic heterogeneity and individual predisposition to cancer.* TRENDS in Genetics 26, 132-141.

[9] G. Hemani at al. 2013. *An evolutionary perspective on epistasis and the missing heritability*. PLoS genetics, e1003295.

[10] L. Xu et al. 2012. *Dynamic epistasis for different alleles of the same gene.* Proceedings of the National Academy of Sciences of the United States of America 109, 10420-10425.

[11] *IMPUTE* Online at: https://mathgen.stats.ox.ac.uk/impute/impute.html (referenced 25/07/2013).

[12] *Haploview* Online at: http://www.broadinstitute.org/scientific-community/science/programs/medical-and-population-genetics/haploview/haploview (referenced 25/07/2013).

[13] *PLINK tutorial*. Online at: http://pngu.mgh.harvard.edu/~purcell/plink/tutorial.shtml (referenced 06/06/2013).

[14] K. Wang et al. 2012, *A novel locus for body mass index on 5p15.2: A meta-analysis of two genome-wide association studies*, Gene 500, 80-84.

[15] C.Hu et al. 2013, *BCL9 and C9orf5 are associated with Negative Symptoms in Schizophrenia: Meta-Analysis of Two Genome-Wide Association Studies*, Plos One 8.

[16] A. Agrawal et al. 2010, *A genome-wide association study of DSM-IV cannabis dependence*, Addict. Biol. 16, 514-518.

[17] F. Deng et al. 2013, *Genome-wide association study identified UQCC locus for spine bone size in humans*, Bone 53, 129-133.

[18] D. Ge et al. 2009, *Genetic variation in IL28B hepatitis C treatment-induced viral clearance*, Nature 461, 399-401.

[19] J.C. Turton et al. 2011, *Investigating Statistical Epistasis in Complex Disorders*, Journal of Alzheimer's Disease 25, 635-644.

[20] J.C. Barrett et al. 2008, *Genome-wide association defines more than 30 distinct susceptibility loci for Crohn's disease.* Nat. Genet. 40, 955-962.

[21] H. Shi et al. 2011, *Using Fisher's method with PLINK 'LD clumped' output to compare SNP effects across Genome-wide Association Study (WGAS)15 datasets*, Int. J. Molec. Epistem. Genet. 2, 30-35.

[22] *Roslin Institute*. Online at: http://www.roslin.ed.ac.uk/ (referenced 17/08/2013).

[23] *Valgrind's DRD tool*. Online at: http://valgrind.org/ (referenced 02/08/2013).

[24] O. Carlborg, C.S. Haley. 2004 *Epistasis: too often neglected in complex trait studies?* Nat. Rev. Genet. 5, 618-625.

[25] O. Zuk et al. 2012. *The mystery of missing heritability: genetic interactions create phantom heritability.* Proceedings of the National Academy of Sciences 109, 1193-1198.

[26] H. Zhao et al. 2003. *Haplotype analysis in population genetics and association studies*. Pharmacogenomics 4, 171-178.