Manycore Algorithms for Genetic Linkage Analysis

Alan John Medlar

A dissertation submitted in partial fulfilment

of the requirements for the degree of

Doctor of Philosophy

of the

University College London.

Division of Medicine

October 2012

I, Alan John Medlar confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Exact algorithms to perform linkage analysis scale exponentially with the size of the input. Beyond a critical point, the amount of work that needs to be done exceeds both available time and memory. In these circumstances, we are forced to either abbreviate the input in some manner or else use an approximation. Approximate methods, like Markov chain Monte Carlo (MCMC), though they make the problem tractable, can take an immense amount of time to converge. The problem of high convergence time is compounded by software which is single-threaded and, as computer processors are manufactured with increasing numbers of physical processing cores, are not designed to take advantage of the available processing power.

In this thesis, we will describe our program *SwiftLink* that embodies our work adapting existing Gibbs samplers to modern computer processor architectures. The processor architectures we target are: multicore processors, that currently feature between 4–8 processor cores, and computer graphics cards (GPUs) that already feature hundreds of processor cores. We implemented parallel versions of the meiosis sampler, that mixes well with tightly linked markers but suffers from irreducibility issues, and the locus sampler which is guaranteed to be irreducible but mixes slowly with tightly linked markers.

We evaluate SwiftLink's performance on real-world datasets of large consanguineous families. We demonstrate that using four processor cores for a single analysis is 3–3.2x faster than the single-threaded implementation of SwiftLink. With respect to the existing MCMC-based programs: it achieves a 6.6–8.7x speedup compared to Morgan and a 66.4–72.3x speedup compared to Simwalk. Utilising both a multicore processor and a GPU concurrently performs 7–7.9x faster than the single-threaded implementation, a 17.6–19x speedup compared to Morgan and a 145.5–192.3x speedup compared to Simwalk.

To my family

Acknowledgements

This thesis began with a poorly worded job advertisement for a position I ended up declining. However, after a few months of consulting, I was hooked! I quit my PhD in wireless networks (I might have been kicked out as well, it was not terrifically clear at the time) and began worrying about genes.

Foremost, I want to acknowledge Robert Kleta and Horia Stanescu, who took many risks working with someone with a different background. Without their patience and infectious enthusiasm for research, I would have given up long ago, let alone still be in the field. Thanks for putting up with me! For his proof reading skills, valuable criticism and friendship, I thank Liam McNamara. I thank Kevin Bryson for being my second supervisor and labmates Anselm, Riko, Naina, Vaksha, Graciana, Detlef, Shazia and everyone in Nephrology for making coming to work anything but dull.

Almost all of my writing up was done in Helsinki as a visiting researcher at HIIT. I fear nothing would have gotten finished without this time to focus. I am thankful for the assistance of Sami Kaski, who made my stay possible, and Petri Myllymäki whose kindness and encyclopedic knowledge of local pubs was very much appreciated by a Brit living in a foreign country.

Lastly, I want to thank Dorota, who has shown more patience and compassion during the last few years than I rightly deserve. I love you.

Helsinki

June 2012

Contents

List of Figures

List of Tables

1 Introduction

> For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers
>
> *Gene Amdahl (1967)*

Herb Sutter's article "The free lunch is over" from the March 2005 issue of Dr Dobb's journal [117], observed that the ongoing trend in computer processors (CPU) had changed from one of ever faster single processors to ever more numerous processors of the same speed. In order for computer programmers to fully exploit the processing power available to them, they would now need to write code that is explicitly parallel. When we say software is "parallel" we mean that it is capable of utilising multiple processors at the same time.

The "free lunch" Sutter referred to is that previously a computer program could be made to run faster simply by being run on a newer processor. This was "free" because it required no intervention from the programmer, i.e. code did not need to be ported to a new architecture, nor did it need to be recompiled. Gains in processor speed traditionally came from concurrency-agnostic means. Processor clock speeds routinely increased as permitted by the miniaturisation of chips; meaning more work is done in the same time period. In addition, the increasing number of cycles were spent more intelligently, for example, pipe-lining multiple instructions at once, speculative execution via branch prediction and out-of-order execution. Computer processors increasingly manipulated the flow of instructions dynamically to better fit their underlying archi-

tecture, improving throughput. Finally, caching was utilised to exploit locality in the executing program, avoiding the need to perform costly main memory accesses in otherwise CPU-bound code. Processor manufacturers cannot continue with this strategy because any gains that could theoretically be provided are overshadowed by physical constraints. Power requirements become too high, heat dissipation is problematic without advanced cooling solutions and increased current leakage affects chip stability.

Computer processors that feature multiple processor cores (multicore processors from now on) continue the long running prediction of Moore's law that total transistor count continues to double approximately every 2 years [1] [87]. Unfortunately, for software to be able to take advantage of the additional processing power, it often needs to be reimplemented and concurrent programming is hard. Programmers are taught how to program by thinking in logical steps, but parallel code is harder to reason about. In concurrent programming there is no deterministic ordering that events are guaranteed to happen in. Concurrent programming requires that the programmer understand where the order of events is important and serialise them explicitly. This can be problematic because in doing so the programmer may overzealously add so much overhead that a multithreaded program runs slower than its serial equivalent. In addition, some applications are just hard to parallelise because of strong dependencies in the ordering of events.

Many of the problems in computational biology and medicine are combinatorial in nature and, as such, would benefit from taking advantage of the increased processing capabilities of multi- and manycore processors. The application that is the focus of this thesis is genetic linkage analysis which was identified as a suitable problem as it is actively used in research and increasingly in clinical settings where it can be a bottleneck in genetic diagnoses. The primary motivation for selecting linkage analysis is essentially practical in nature. Our lab is involved in a bioinformatics core facility,

[1] The figure of 18 months is often attributed to Moore, however it was David House of Intel who stated in 1975 that performance would double in that time. This is due to the factors we have stated, other than transistor count, that affect processor speed [50].

related to the Institute of Child Health, where we provide expertise in genetic linkage analysis to a wide group of institutes and clinicians. The service is provided to those for whom this kind of analysis is not their main expertise and as an additional vector to filter the results of next-generation sequencing projects focused on disease. We noticed that whilst a majority of projects could be performed by programs that use existing exact algorithms, the remaining larger projects were problematic due to excessive run-times.

Linkage analysis is a statistical procedure that aims to map the location of a given trait to a genetic locus. It does this by inferring historic meiotic events in a family based on the genotypes of a subset of the individuals. Exact algorithms exist to evaluate this likelihood, but they all have their drawbacks. The two main families of algorithm are the Elston-Stewart algorithm and the Lander-Green algorithm. The Elston-Stewart algorithm can handle arbitrarily large and complex families, but does not scale beyond a few markers. The Lander-Green algorithm can handle a large number of markers, for example, spanning an entire chromosome, but is restricted to small-to-medium sized families. All exact algorithms quickly become memory-bound due to the number of intermediate results that need to be stored. Parallel processing can only aid in CPU-bound problems, so barring an algorithmic improvement, multicore cannot be applied to great effect. If a problem goes beyond a critical limit (this limit being data dependent) then the analysis must be approximated instead.

There are numerous approximations that can be applied. The first class of approximations still use the same exact algorithms, but abbreviate the input data in some manner, dependent on how the problem scales. For example, if we are performing a genome-wide linkage scan, then it is likely we are using the Lander-Green algorithm. If there are many unaffected children in the last generation of a pedigree with a recessive phenotype, we could remove them one at a time until the problem can be run given the amount of computer memory and time we have. One downside of this is that we do not know what the optimal configuration of individuals is. We could run all combinations, but each may give a potentially contradictory result. Indeed, in the worst case scenario, this might hide the fact someone has been misdiagnosed. Another common approach is

to run short windows of markers across the genome. This will be easier to run than the entire chromosome, but may result in errors where the windows meet.

A more systematic approach is to run an algorithmic approximation without abbreviating the input data at all. Stochastic simulations of linkage analysis have been shown, where possible, to be highly accurate [126] and avoid many of the problems described above. The main downside is that they are slow to converge to an answer. Recent years have seen an explosion in the availability of marker data, so whilst the potential work load has increased, processor speed has not. As the work is CPU-bound, it is possible that redesigning these particular algorithms could result in a big win, making analysis faster and easier.

For stochastic simulations, it is not immediately obvious how they would run in parallel. They tend to be based on Markov chain Monte Carlo (MCMC), where the states of the chain form a strict sequential ordering. Our ability to parallelise MCMC is highly problem specific. Different ways this has been done in the past include: parallelising expensive likelihood calculations [116], pre-fetching multiple next steps in the chain in parallel [11], multiple Markov chains running in parallel on the complete state space [72] and partitioning the state space with a single Markov chain for each partition [13]. However, it is not obvious what is the best payoff for a given problem. In this thesis, we do not present a complete analysis of different parallelism schemes as exhaustively covering all options would not be possible. Instead, we take an existing set of Gibbs samplers and adapt them to two very different parallel architectures. This was achieved by both trying to retain the serial ordering of events where possible and by using techniques that afforded the most scalability with a view to running on higher numbers of processors in the future.

The two parallel architectures we focus on are multicore processors and computer graphics cards. Multicore processors now routinely contain four processor cores and are starting to be released with six and eight cores. In addition, Intel CPUs can be enabled with hyperthreading, which from the perspective of the operating system appear as two

logical cores per physical core. In the near future, multicore processors are predicted to feature many tens to hundreds of cores [82]. The second architecture we focus on, graphics cards (GPU), have in the last decade transitioned from hardware designed to accelerate computer graphics to more general purpose architectures like that of a regular CPU. GPUs already feature hundreds of processor cores and, as such, will be used to test the limitations of our work at scale. GPUs in recent years have become general purpose enough to be used extensively in statistical computing [72], protein folding [26], phylogenetics [116], sequence alignment [125, 78] and are starting to be seen even in small compute clusters. Of course, there is the caveat that future CPU architectures will no doubt differ considerably from current GPUs, however we will use them to better understand how future processor architectures can be better exploited by statistical computing applications.

1.1 Contributions

This thesis details several designs for parallel Gibbs samplers for multipoint genetic linkage analysis and presents the corresponding software implementation: *SwiftLink*. SwiftLink's sampler implementations are each focused on specific hardware platforms and we detail how each platform influenced the design. In most cases, this affects the accuracy of the results benignly, but in others detrimentally.

Multicore linkage analysis

We show how simple modifications to existing Gibbs samplers can produce parallel implementations that retain the same level of accuracy and scale beyond the number of processor cores currently available.

GPU linkage analysis

We show that given the extensive modifications necessary to write a GPU application capable of executing over hundreds of cores, we can achieve large speed improvements for several of the modules necessary to perform linkage analysis.

We show that the degree of speedup is closely related to how well the algorithm fits the actual hardware architecture. If it does not fit the architecture well, then there is a trade-off between the accuracy of the results and the level of hardware utilisation.

Heterogeneous GPU / multicore linkage analysis

We demonstrate it is possible to utilise the strengths of both multicore CPU and GPU architectures in different aspects of the same program. This produces a faster analysis than using either GPU or CPU alone, with a similar level of accuracy compared to the multicore program.

Evaluation on real-world datasets

We evaluate SwiftLink on real-world datasets of both small and large families, genotyped with many markers. A small family is evaluated with an exact algorithm, the results of which are used to assess the accuracy of our program. Large families with many markers are beyond the reach of exact linkage algorithms, so these datasets can only be compared with other MCMC-based applications. We show that SwiftLink performs a complete genome-wide linkage scan as much as 19x faster compared to an existing single-threaded implementation of the same samplers.

1.2 Thesis Outline

The thesis proceeds in a bottom-up manner, first looking at background material, designing the parallel Gibbs samplers and then evaluating their efficacy.

Molecular genetics background

In Chapter 2, we will cover the background information in molecular genetics necessary to understand the context of the analysis and give a rationale for many of the details for why inheritance is modelled the way it is. We will look at DNA, chromosomes, the process of meiosis and how this produces the classical

segregation patterns found in Mendelian traits, such as the monogenic diseases we will be investigating.

Genetic linkage analysis

In Chapter 3, we introduce the concept of genetic linkage analysis and provide the theoretical background for the mapping of disease traits. The chapter proceeds to describe the main techniques that have been employed to solve the necessary likelihood calculations and how this was motivated by the development of ever numerous molecular markers.

Statistical methods for linkage analysis

Chapter 4 contains a formal description of the Gibbs samplers used throughout this thesis. It covers the whole locus Gibbs sampler, the whole meiosis Gibbs sampler and all of the relevant algorithms necessary to evaluate them.

Parallel sampler design

Chapter 5 will give a broad overview of both the different hardware and software that is available to us, with an emphasis on how they will affect the design of our parallel samplers. The chapter concludes by sketching the basic design of SwiftLink, highlighting any parameters that need to be investigated.

Software implementation

Our software implementation of SwiftLink is described in Chapter 6. There are many details about the design of the parallel samplers, identified in the previous chapter, that must be investigated empirically. We take each issue in turn and perform experiments to better understand the hardware and programming paradigms we are using, with a view to maximising performance. Where we are unable to maintain the serial ordering of events, approximations are identified and their effect on results investigated.

Case studies

Whilst we have optimised our software for both multicore CPUs and manycore

GPUs, the benchmarks from the previous chapter do not give a clear picture of what the improved performance means for an actual linkage project nor how it compares with currently available software. In Chapter 7, we will take several complete projects and evaluate SwiftLink in terms of both accuracy and run-time.

Conclusions

We conclude in Chapter 8 by providing a critical evaluation of our work and the lessons that were learnt for the field. Finally we outline the directions future research might take from here.

1.3 Publications

A publication resulting from the work contained in this thesis is currently in preparation. Linkage analyses for sensorineural deafness and benign chorea pedigrees in Chapter 7 will form part of future publications as well.

SwiftLink: Parallel MCMC linkage analysis utilising multicore CPU and GPU
Medlar A, Głowacka D, Stanescu H, Bryson K, Kleta R

2 Molecular Genetics Background

I can't be as confident about computer science as I can about biology.
Biology easily has 500 years of exciting problems to work on.

Donald E. Knuth

This chapter is aimed towards those who do not have a background in biology, genetics or medicine. The goal is to give the necessary context and vocabulary to understand the rest of the thesis and can safely be skipped by those already familiar with the topics. We take a bottom-up approach, first looking at DNA and building up to the mechanisms of inheritance.

2.1 DNA

DNA (deoxyribonucleic acid) is a linear molecule composed of *nucleotides*, the combination of which, encode the genetic information that constitutes the genome of the host organism. DNA is packaged into structures called *chromosomes*, found in the nucleus of eukaryotic (all organisms that are not bacteria or archea) cells.

2.1.1 Structure

Each DNA molecule is made from two long strands of nucleotides connected via hydrogen bonds in a double helix. Each nucleotide contains a molecule of 2-deoxyribose, a phosphate group and a nucleobase. The alternating sugar and phosphate groups form

two backbones on either side of the bases being held. The possible bases are adenine, guanine, thymine and cytosine (A, G, T and C, respectively). All adenine bases are paired with thymine and all guanine bases are paired with cytosine. The combinations of AT and GC on opposite strands are referred to as *base pairs* (bp). The complete human reference genome contains approximately 3.3 billion base pairs of DNA.

2.1.2 Function

All of the proteins that make up our bodies are encoded in DNA as *genes*. Proteins are composed of combinations of amino acids arranged in a linear sequence. These linear sequences of amino acids fold into three dimensional structures, the conformation of which dictates their function. Each amino acid is encoded in DNA by a triplet of nucleotides called a *codon*. There is inherent redundancy within the genetic code, as the 64 different codons map to only 20 amino acids.

There are special codons within DNA called *start codons* and *stop codons* that delineate the extent of a gene. The entire gene is *transcribed* into a molecule called messenger ribonucleic acid (mRNA). The mRNA is essentially a copy of the transcribed DNA, but with a few differences. The sugar is ribose instead of deoxyribose and instead of thymine, RNA uses a base called uracil (U). The mRNA then undergoes *splicing* to remove all non-coding regions from the sequence. The spliced mRNA is *translated* to a protein by a ribosome.

The non-coding regions within a gene are called *introns*, contrasted with the coding regions that encode the sequence of amino acids, called *exons*. A single gene can be made from many exons and may have several splice variants, leading to the creation of different proteins.

2.1.3 Nomenclature

Each strand of DNA has a polarity, one end is referred to as the $3'$ end and the other as the $5'$ end. The numbering comes from the orientation of the carbon atoms in the deoxyribose molecule (numbered $1'$ to $5'$). The $3'$ end is terminated with the hydroxyl

Figure 2.1: Karyogram for a male showing 23 pairs of chromosomes (image credit: NHGRI [92]).

group of a sugar and the $5'$ end is terminated with a phosphate group. The two strands lie antiparallel to one another, namely the strands sit in opposite directions.

A DNA sequence is by convention the linear sequence of nucleotides that sit on the $5'$ to $3'$ strand, also called the *sense* or *forward* strand. The opposing strand ($3'$ to $5'$, *antisense* or *reverse* strand) contains the complement sequence.

2.2 Chromosomes

DNA is packaged along with specialised proteins into structures called chromosomes. The proteins involved provide structure, regulation and perform the reading, replication and repair of DNA. The exact number of chromosomes will depend on the type of cell. *Haploid* cells, such as unfertilised eggs and sperm, contain a single copy of each of the 23 human chromosomes, whereas *diploid* cells contain two copies. Each diploid cell in the body will contain that person's complete genome.

The chromosomes that make up a pair are called *homologs*. Of the 23 chromosome pairs, 1 to 22 are *autosomal* chromosomes and the 23^{rd} pair are sex-determining chromosomes. The sex-determining chromosomes are called the X and Y chromosomes. Females have a homologous pair of X chromosomes, whereas males have an X and a Y chromosome.

During the final stages of cell division, we can artificially halt the process to inspect

individual chromosomes. Figure 2.1 shows a *karyogram* which features a complete set of arranged and stained chromosomes. Each chromosome has a short *p* arm (petit) and longer *q* arm (queue), connected by the *centromere*. Both ends of each chromosome have a *telomere*. The banding pattern from staining permits us to distinguish different regions of the chromosome. Bands are identified like 1q23.2, which is on chromosome 1, q arm, region 2, band 3, sub-band 2.

Each pair of chromosomes will feature one complete set of genes [1]. The two copies of each gene are referred to as *alleles* of that gene. The term allele can describe an arbitrary genetic locus, so is not limited to genes.

2.3 Heredity

Traits, or *phenotypes*, are passed between generations by the fusing of genetic material from both mother and father. The immense variability we see in people around us is due to the process by which DNA is assorted called *meiosis*. Meiosis is a kind of cell division that takes diploid cells and turns them into haploid germ cells or *gametes*. Gametes, one from each parent, combine to form a diploid *zygote* containing information from both parents.

2.3.1 Meiosis

Meiosis is the process by which a single diploid cell will subdivide into four distinct haploid cells. Meiosis consists of three distinct phases: the *meiotic S phase*, *meiosis I* and *meiosis II*, see Figure 2.2.

Meiotic S Phase

During the meiotic S phase, the paternal and maternal homologs of each chromosome pair replicate themselves completely. The cell now contains twice the original number of chromosomes.

[1] Unless any of those genes have been deleted or duplicated.

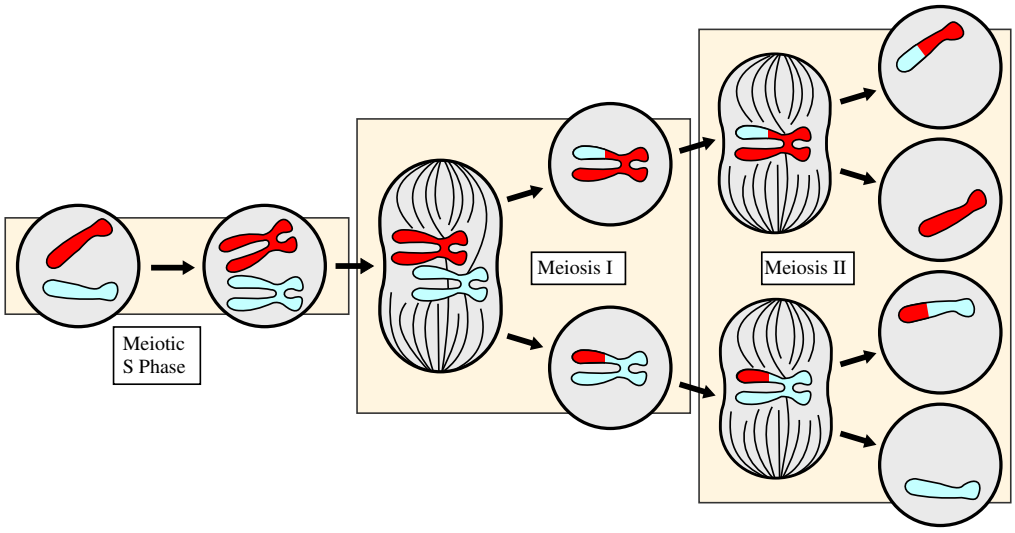


Figure 2.2: Simplified view of meiosis showing a single pair of chromosomes throughout each phase of the process (image credit: NCBI and Wikimedia Commons, modified [79]).

Meiosis I

In meiosis I, both maternal and paternal homologs pair up forming a *bivalent* consisting of four chromosomes. During cell division, for each bivalent, the maternal homologs will be pulled to one end of the cell and the paternal homologs to the other. For different chromosomes this is independent, for example, both maternal chromosome 1 copies do not necessarily get pulled to the same end as both maternal chromosome 2 copies. Meiosis I produces two diploid daughter cells each with a complement of 23 chromosome pairs.

Meiosis II

Meiosis II is identical to the cell division in meiosis I, except that this time the result of the cell division is two haploid gametes from each of the two daughter cells from meiosis I.

2.3.2 Recombination

During meiosis I, maternal and paternal homologs align with one another to form bivalents. These bivalents are held together with connections called *chiasmata*. Recombination, or crossing-over, occurs at chiasma where DNA is broken and reattached in

such a way that chromosomes within the bivalent exchange genetic material. This process is indicated by the bivalents in Figure 2.2 becoming multi-coloured, as they have exchanged the whole q arm by the second half of meiosis I.

Between any two points on the genome there is a probability that a recombination will occur. The recombination probability increases the further apart the two locations are, up to the value 0.5 which states that the two locations segregate independently (which is the case when two loci are on different chromosomes). Recombination probabilities are not constant for a given length of DNA as there are numerous hot and coldspots of activity. In addition, recombination probabilities tend to be higher in women, who have approximately 60% more recombination events compared to men [12, 55, 16].

The recombination events that occur during meiosis, the gametes of which go on to produce the next generation of individuals, are the key to understanding how we inherit traits from our parents. Locating the locus on a chromosome that is responsible for a trait is a process called *mapping*. In order to map traits that we care about, they must be mapped relative to something, therefore we need genetic sign-posts or *markers* that are relatively constant. Classically, this would have been done by conducting breeding experiments and assessing the amount different traits appear together, mapping them in relation to one another. Nowadays we use molecular markers made from DNA.

2.4 Molecular Markers

Molecular markers are variations within the DNA sequence itself of an organism. To be useful for the purposes of mapping they must have the following properties:

Known location

Mapping exercises involve us taking something of unknown location (for example, a disease trait) and calculating the likelihood that it is located close to different markers of known location.

Co-dominance

Co-dominance implies that we are able to clearly ascertain all types of a given marker, i.e. all possible values it could take can be distinguished from one another.

Polymorphic

There must be some degree of variation within the population from which to infer how material was inherited. We refer to each possible form of a marker as an *allele*. If there are two possible values, then the marker is said to be *biallelic*[2] (and three values, triallelic etc). If all instances of a marker were all the same, then all modes of inheritance will be equally likely.

Low mutation rate

We want to be sure that the marker typed in an individual was genuinely inherited from one of their parents. Some mutations are simple to detect, for example, if a biallelic marker becomes triallelic, but other mutations may be silent.

Hardy-Weinburg Equilibrium

Hardy-Weinburg Equilibrium is the ratio of different genotype frequencies resultant from the assumption of random mating. Namely, a biallelic marker with minor allele frequency p and $q = (1 - p)$ will produce genotypes in the ratio $p^2 : 2pq : q^2$.

The two most commonly used molecular markers today are *short tandem repeats* and *single nucleotide polymorphisms*.

2.4.1 Short Tandem Repeats

Short tandem repeats (STR), also known as microsatellites, are molecular markers where a short stretch of DNA (typically 1-3 bp) is repeated several times, contiguously. They are highly polymorphic and therefore informative for mapping. STRs are found almost exclusively in non-coding regions, due to the impact they would have on

[2]Curiously not called *di-* allelic, though in this thesis we will stick to conventions.

gene reading frames. STRs have largely been superseded by single nucleotide polymorphisms for reasons of cost and higher density.

2.4.2 Single Nucleotide Polymorphisms

Single nucleotide polymorphisms (SNP) are variations found in a single base pair of DNA. Whilst a given SNP could have four possible values (A, T, G or C), a large proportion of them are biallelic. This lower information content, compared to STR markers, is made up for by their immense density. The human genome contains a SNP approximately every 300 base pairs.

The strict definition of what constitutes a SNP places a lower bound of 1% on its minor allele frequency, however, this is a historic definition. Current databases of variation include SNPs at far lower frequencies. This is permissible because of the high accuracy of genotyping and large sample sizes used.

2.4.3 Marker Maps

One of the important properties of a marker is that we know its location relative to other markers. This will allow us to *map* genes, polymorphisms and disease traits to specific locations on different chromosomes. There are several types of map that we can use to identify the location of a molecular marker, the main two being *physical maps* and *genetic maps*.

Physical Maps

Physical maps localise points on the genome to an individual base pair resolution in the actual DNA sequence itself. An individual base pair of DNA in the human reference genome is identified by the chromosome and the offset with respect to the start of that chromosome. When stating the location of a mutation in the genome, we would use a physical map.

Genetic Maps

Genetic maps are a little different from physical maps. The distances between markers are measured by their genetic distance in centiMorgans (cM). 1 cM is equal to a 0.01 chance of a recombination. Whilst this is often proportional to the physical distance, the physical distance cannot express, for example, the differences between male and female recombination rates. Knowing genetic distances permits an accurate probabilistic analysis of how genetic material flows from generation to generation and we will make extensive use of it in the remainder of this thesis.

2.5 Mendelian Inheritance

We are only interested in discrete, binary traits in this thesis. We will ignore other continuous and age-of-onset phenotypes. For a binary trait, we need to define the probability that an individual will have the disease phenotype, given the disease alleles that were inherited from both parents. The two possible alleles are d for the normal or wild-type and D for the mutant. Therefore, if we know a person's disease status, $\mathbf{Y}$, and we know what disease alleles were inherited, G (where G can be dd, dD or DD), then we must estimate $P(\mathbf{Y} \mid G)$. We refer to this as the *penetrance function*.

For simple Mendelian disorders, we identify the correct penetrance function by assessing which of the classical *segregation patterns* corresponds to the observed pattern of disease in a given family. Before we enumerate the classical segregation patterns, we will look at the Mendelian laws of inheritance and state how a pedigree diagram is interpreted.

2.5.1 Mendel's Laws

From observations made during cross breeding experiments with garden peas, Gregor Mendel observed common features of how traits segregate through different generations. These observations were later formalised as Mendel's laws. They are as follows:

Law of Segregation:

Mendel found that if two homozygous versions of the *same* trait (tt and TT) are cross-bred, then the next generation will contain only heterozygous (tT) offspring. If we cross-breed the new offspring with one another, we will get the ratio 1:2:1 for tt, tT and TT, respectively.

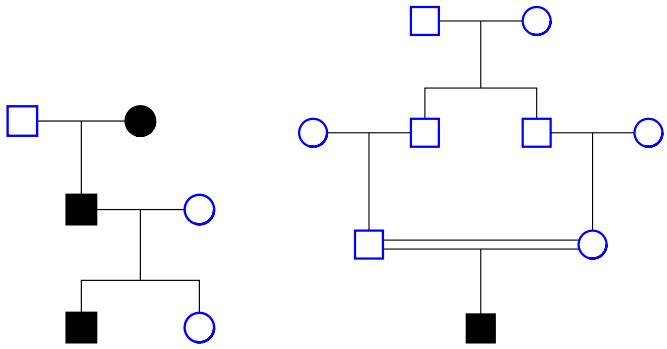
The law of segregation states that each individual has two copies of a trait (one on each chromosome in a given pair) and we are equally likely to pass on one or the other to one of our offspring. This is because zygotes are composed of two gametes, both of which are haploid. One gamete is provided by each parent and only contains half that parent's chromosomes.

Law of Independent Assortment:

The law of independent assortment states that two *different* traits will segregate independently. We know now that this is actually only true in the special case where those traits are located on different chromosomes. Otherwise those two traits will co-segregate to some degree relative to the genetic distance separating them.

2.5.2 Pedigrees

Familial relationships are represented diagrammatically with a *pedigree*. Pedigrees are graphs that specify individuals' relationships to one another. Males are represented by squares and females with circles. A pedigree contains two types of individual: *founders*, those whose parents are not in the pedigree and *non-founders*, everyone else. If the pedigree is inbred, then the graph will be cyclic and is indicated by a double line connecting the related individuals (e.g. Figure 2.3b). We call this a *consanguineous pedigree*. A pedigree without any inbreeding is called an *outbred pedigree* (e.g. Figure 2.3a). A pedigree must state the *affection status* for all members of the pedigree. This is a statement of whether each person has the phenotype or not. There are 3 possible values: affected (coloured black), unaffected (white) and unknown (grey).



(a) Autosomal Dominant (b) Autosomal Recessive

Figure 2.3: Segregation patterns for autosomal dominant and autosomal recessive disease models. Autosomal dominant phenotypes are expressed by individuals who carry at least one copy of the disease trait, whereas autosomal recessive phenotypes are only expressed by individuals with two copies. Double lines indicate an inbreeding loop.

The complexity of a pedigree is referred to as the bit size. The bit size is equal to $2n - f$, where n is the number of non-founders and f is the number of founders. The bit size reflects the number of meiosis ($2n$) within the pedigree and that founder meioses are uninformative because their parents are not present by definition (hence $-f$).

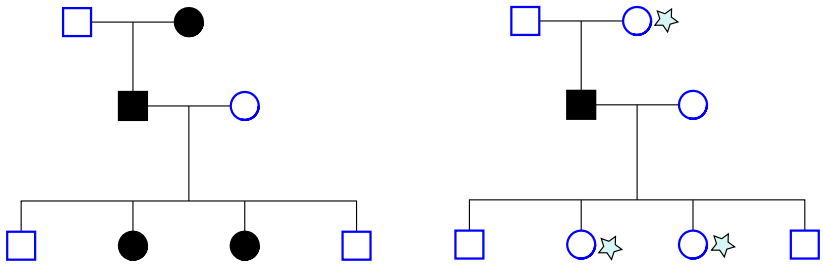
2.5.3 Segregation Patterns

The classical patterns of Mendelian inheritance are special cases that are *fully penetrant*, i.e. $P(\mathbf{Y} = \textit{affected} \mid G)$ is either 0 or 1.

Autosomal Dominant

Autosomal dominant diseases are where the disease trait is located on one of the 22 autosomal chromosomes and individuals require at least one copy of the disease allele to manifest the disease, therefore $P(\mathbf{Y} \mid dD) = 1$ and $P(\mathbf{Y} \mid DD) = 1$.

Figure 2.3a shows a pedigree with an autosomal dominant disease. Every affected individuals will, by definition, have at least one affected parent. Therefore all generations will contain affected individuals. As the disease trait is found on an autosome, males and females are equally likely to be affected and can both transmit the disease trait to their offspring. Finally, as the disease only requires a single copy of the disease allele, if an individual has two unaffected parents, then they cannot have the disease



(a) X-linked Dominant (b) X-linked Recessive

Figure 2.4: Segregation patterns for X-linked dominant and X-linked recessive disease models. Carriers in the X-linked recessive pedigree have been indicated with stars. The key difference in X-linked phenotypes is how many X chromosomes there are in total. So, for example, males only require a single copy of the disease trait to manifest an X-linked recessive disease as they have only a single X chromosome.

themselves.

Autosomal Recessive

Autosomal recessive diseases are where the disease trait is located on one of the 22 autosomal chromosomes and individuals need two copies of the disease allele to have the disease, ($P(\mathbf{Y} \mid DD) = 1$).

Figure 2.3b shows a pedigree with an autosomal recessive disease. Unlike an autosomal dominant disease, affected individuals can be the offspring of two unaffected parents. Whilst this is not shown in our example, the disease can appear to skip generations, the unaffected intermediate generations only being carriers. The disease is equally likely in both males and females, again because it is located on an autosome. Where two parents are both affected, all offspring must also be affected. Recessive disease are more likely in consanguineous (inbred) families.

X-linked Dominant

The disease trait can be located on the X chromosome. As males and females do not have the same number of X chromosomes, the segregation pattern will appear different from autosomal dominant diseases.

In an X-linked dominant disease, if the father is affected, then all of his daughters will be affected. Sons can only inherit their X chromosome from their mother, so males

need an affected mother to have any chance of being affected (50% if the mother has one disease allele, 100% if she has two copies). An example is shown in Figure 2.4a.

X-linked Recessive

X-linked recessive diseases need all X chromosomes to have the disease allele in an individual. For a daughter to be affected, her father must be affected and her mother be at least a carrier. Sons cannot inherit the X chromosome from the father, so need a carrier mother to be affected, so it may appear to skip generations if the mother is not affected herself. See Figure 2.4b for an example; all of the individuals that are carriers are indicated with a star.

2.6 Disease Mapping

Disease mapping is the process by which we identify genomic regions that either contain causative mutations or risk-enhancing variants for a given phenotype. Diseases are often categorised into "simple" Mendelian traits that segregate through a pedigree with an identifiable pattern (described in Section 2.5.3) and "complex" traits, that, whilst there is a clear genetic component (i.e. it has been shown to run in families), there is no Mendelian segregation pattern that adequately describes its transmission. Whereas Mendelian traits tend to be caused by mutations in single genes (monogenic disorders), complex traits are multifactorial, influenced by many genes and other non-genetic factors, e.g. environment.

In this section we will give a brief overview of linkage analysis, association analysis and high-throughput sequencing technologies.

2.6.1 Linkage Analysis

Parametric (or model-based) linkage analysis aims to identify an explicit relationship between the phenotype and the transmission of chromosomal regions through families containing affected individuals. The disease trait is mapped by assessing the extent

to which the disease trait cosegregates (is inherited together) with markers of known location, given a genetic model. The genetic model specifies the penetrance function of the trait together with population allele frequencies. A description of how multipoint parametric linkage analysis is performed can be found in the next chapter.

Parametric linkage analysis has one main disadvantage: it relies on our ability to ensure the correct genetic model is specified. In the case of traits without a clear-cut segregation pattern, these penetrance values must be estimated based on a representative sample of many, often small, families, which may not be informative enough to identify more complex penetrance functions, for example, including variable age-of-onset and other factors.

Nonparametric (or model-free) linkage analysis does not depend on a genetic model for the disease locus and instead measures the level of allele sharing between affected sib-pairs. Nonparametric analysis is less powerful than parametric methods and requires several families to be collected, whereas parametric analysis may require only one informative pedigree. However, as there is no need to define a genetic model, nonparametric analysis can be more robust.

Both parametric and nonparametric linkage analysis can fail to find the disease locus if the phenotype has genetic heterogeneity, that is, despite the disease trait having the same transmission pattern, single mutations in different genes cause the same (or what appears to be the same) phenotype in different families.

2.6.2 Association Analysis

Analysis of common, complex genetic traits [63], for which there is no clear segregation pattern, can be investigated in an *association study*. Association studies can be used to investigate a single candidate polymorphism, the polymorphisms within one or more candidate genes or genome-wide (referred to a genome-wide association study or GWAS). Whereas candidate gene studies may only type a handful of SNPs, GWAS will routinely use $> 1,000,000$ SNPs thanks to high-throughput genotyping. Genetic association studies aim to identify correlations between the disease status of unrelated

(or at least distantly related) individuals and the alleles typed at genetic markers. A genetic association exists if specific alleles are found to occur more frequently in cases compared to controls than would be expected by chance. The size of cohort necessary to achieve significance is dependant on our ability to define the phenotype. Whilst the causes of rarer diseases (with severe phenotypes) can be identified with only a few hundred patients (e.g. [53, 114]), common traits investigated with GWAS can involve thousands of patients.

Care must be taken to minimise the possibility of spurious associations. In addition to removing poor quality data evidenced by low genotype call-rates, several quality controls should be performed prior to analysis:

Hardy–Weinburg equilibrium:

SNPs that deviate from Hardy-Weinburg equilibrium (briefly described in Section 2.4) must be removed from the dataset as these can be an indication of mistyped SNPs or the result of indels. Population stratification, inbreeding and non-random mating are other possible causes of a deviation from Hardy-Weinburg equilibrium.

Population stratification:

Both cases and controls that make up the cohort must come from a homogeneous population as any differences in population structure may induce false-positive association signals in the final analysis. Population stratification can be assessed by clustering individuals with principle component analysis (PCA) and the resulting plot visually inspected to remove any outliers.

Single SNP associations are commonly assessed with χ^2 tests, but many other test statistics, for example, the Cochran–Armitage test [5], are in common use. As many tests are performed, the significance level must be adjusted for multiple comparisons, for example, by Bonferroni correction [9]. Genetic associations can be caused for different reasons:

Direct causal association:

If a typed genetic marker is the disease locus itself, then the association is due to the direct relationship between the causative mutation and the disease trait. This will typically not be the case for GWAS as they are performed with SNP chips that only feature common polymorphisms, but for a candidate gene study the causative mutation might indeed have been typed.

Indirect association:

Indirect associations are where the association between the disease trait and a specific marker allele is due to that genetic marker being close to the disease locus. In this case, the association is due to *linkage disequilibrium* (LD) between the disease locus and the associated allele.

Linkage disequilibrium is a measure of statistical association between two loci, which can be due to linkage in tightly-linked loci that are frequently co-inherited, or at unlinked loci as a result of selection or non-random mating. Two locus pair-wise measures of linkage disequilibrium are D' and r^2, both of which are commonly used in GWAS, though r^2 is more robust to the presence of rare alleles [24]. Significant association signals are super-imposed on genome-wide LD maps provided by the HapMap project [18]. The locus defined by the LD block containing a significant association tells us which genes should be investigated further by fine-mapping or sequencing. The list of genes can be prioritised by biological relevance if the genes are of known function.

It is important that the results from a given GWAS is replicated in a different population to ensure that the association signals are not due to the structure of the population investigated or any other effects that are difficult to control for. Ultimately, though, the final test of an association is the detection of the causal variant (or variants) and a biological validation determining the mechanism of risk-enhancement.

2.6.3 Next-generation Sequencing

Next-generation sequencing differs from traditional Sanger sequencing [106] in that it is massively parallelised and far cheaper per base pair sequenced. The reduction in run-time and expensive has meant that NGS techniques are starting to be applied throughout the life sciences, not just clinical studies.

Next-generation sequencers tend to produce shorter reads than previously available sequencing technologies, but orders of magnitude more data [3]. The enormous quantity of data generated means we must rely on bioinformatics techniques to automate the process assembling the data into something meaningful, like an entire genome or exome.

Whole Exome Sequencing

Whole exome capture and next-generation sequencing is the targeted sequencing of all (or at least a majority of) protein coding sequence in the human genome and has become a powerful tool to investigate all of the genes contained in a region of interest identified by genetic linkage analysis, for example, in the case where the region contains 100s of genes, this would be prohibitively time-consuming to tackle with Sanger sequencing [7]. Exome sequencing is a suitable strategy as we are, in general, looking for rare variants in the coding sequence or splice sites of a gene leading to a functional consequence, for example, a non-synonymous coding variant or creation of a premature stop codon.

A common approach is to align the exome data to the human reference sequence and perform variant calling. The list of variants is annotated with structural information to assess potential consequences. Known variants from public databases (e.g. dbSNP) are filtered out and the remaining novel variants assessed for their severity. In the best-case scenario, this procedure will leave at least one highly disruptive variant in a gene of biological relevance.

[3] Though, at the time of writing, the Pacific Biosciences RS single molecule sequencer is one of the exceptions, producing far fewer reads than other NGS platforms, but each read can be thousands of base pairs long.

Sequencing the whole exome may seem excessive to investigate a set of perhaps < 100 genes, but economies of scale have made the process cheaper than targeting a specific region of interest and has the added advantage that the data can be generated before the completion of the linkage study.

3 Genetic Linkage Analysis

Two years work wasted. I have been breeding those flies for all that time and I've got nothing out of it!

Thomas Hunt Morgan

This chapter focuses on linkage analysis and its application in mapping disease genes. We will cover the theoretical background that tells us how to calculate the likelihood that two traits are in linkage. We will then survey the major techniques that have been employed over the years to compute this likelihood. Throughout, we hope to give the historic context that led to the development of different methodologies.

3.1 Theoretical Background

Linkage analysis is the process by which we determine the location of a disease trait by looking at the degree to which that trait cosegratates (is inherited together) with markers of known location. If a disease trait cosegregates frequently with another locus, they are said to be *linked* [88] and, therefore, proximate to one another [115]. In the case of parametric linkage analysis, linkage between the disease trait and the genomic sequence is assessed given an explicit genetic model. The genetic model states how the disease is inherited (e.g. autosomal recessive) through a penetrance function and the frequency of the disease allele in the general population. The standard manner of displaying results is as a likelihood ratio called a logarithm of odds (LOD) score [90], comparing the likelihood of two traits being linked versus otherwise.

Traditionally, linkage is assessed in non-humans by directly counting recombinations in controlled crosses, but for humans this is impractical. Instead, we use a series of genetic markers, ideally uniformly spaced along the genome, to infer past meiotic events within a pedigree. This is a computational challenge because we observe the underlying genetic sequence imperfectly due to genotypes lacking phase information (i.e. which homologous chromosome each allele comes from) and non-typed individuals whose genotypes must be inferred using the genotypes of others and the underlying family structure. The consequence being that many possible DNA sequences can correspond to the same observed genotypes and we must consider many individuals and markers jointly in order to ascertain the most likely possibilities.

In general, the assessment of linkage involves calculating the following joint likelihood [100]:

$$\mathcal{L} = \sum_{g_1} \cdots \sum_{g_n} \prod_{i} P(Y_i \mid g_i) \prod_{j} P(g_j) \prod_{\{k,l,m\}} P(g_m \mid g_k, g_l)$$

where g_n and Y_n are the genotypes and phenotypic traits of the n^{th} individual, respectively. Of the three main components, the first is the *penetrance probability* ($P(Y_i \mid g_i)$), which states the probability of manifesting a specific phenotype, given the genotype. The second component is the *founder probability* ($P(g_j)$), which only applies to founders. Given the lack of parental information, we model founder genotypes on the population as a whole. Founder genotypes are assumed to be in Hardy-Weinberg equilibrium and linkage equilibrium. The final component is the *transmission probability* ($P(g_m \mid g_k, g_l)$), which states the probability of a non-founder's genotypes given the genotypes of their parents.

To calculate the likelihood of two traits (e.g. a genetic marker and a disease trait) being linked, we not only need to perform this summation across markers, but also include a term to take into account the likelihood of genotype assignments given those at neighbouring loci. We use Haldane's model [40] as this does not model crossover interference (the process by which that act of a crossover occurring prevents another

crossover from occurring nearby), greatly reducing the necessary computation. Haldane's model involves the use of a genetic map enabling us to calculate the recombination fraction between pairs of markers. A recombination fraction is the probability of an odd number of crossover events occurring in-between the two markers (as an even number of recombinations will appear the same as zero recombinations). The Haldane map function is:

$$\theta = \frac{1}{2}(1 - e^{-2d})$$

where θ is the recombination fraction and d is the genetic distance in Morgans. θ and its inverse can be applied at all informative meioses, i.e. meioses from heterozygous parents.

The LOD score is calculated as the ratio between the likelihood of the disease trait and a marker being linked at a recombination fraction $\hat{\theta}$ and the likelihood of the two traits being unlinked, i.e. segregating independently on different chromosomes. By convention, the LOD score is the log_{10} of the likelihood ratio:

$$LOD(\theta) = log_{10}\left(\frac{\mathcal{L}(\hat{\theta})}{\mathcal{L}(0.5)}\right)$$

A LOD score of -2 or lower is considered enough evidence to exclude that region from linkage. Exclusion mapping can make even a weak pedigree useful. Even if it cannot direct us towards a single locus for further investigation, it will permit us to exclude a large proportion of the genome. Conversely, a score of 3 is considered to have genome-wide significance and historically considered evidence of linkage (for the X-chromosome, 2 is considered significant) [90]. In a genome-wide, multipoint setting, 3.3 is considered significant [61].

Direct calculation of the complete likelihood is intractable for all but the simplest pedigrees. The complexity of a naïve evaluation of all possible combinations of genotypes scales exponentially with both the number of individuals in the pedigree and the number of markers considered jointly.

The analytical methodology employed has changed with technological advances. Initially, only a few phenotypic markers were available for analysis, so it was only important for algorithms to handle many individuals connected by perhaps complex relationships. After the landmark paper by Botstein *et al*. [10], common genetic polymorphisms were seen to be more numerous, providing both more context and improved resolution. This provoked efforts to scale to larger numbers of markers, but as we will see, limited analysis to smaller pedigrees. Work in the statistics literature provided means to bridge the gap between these two extremes. All of these techniques have been adapted over time to handle both new technological advancements in the genetics field, e.g. SNPs, and many have enjoyed being extended to run with new software tricks and hardware architectures.

3.1.1 Assumptions

Irrespective of the algorithm or software employed, in general, linkage analysis is based on the following assumptions:

Linkage equilibrium between markers

The assumption of linkage equilibrium between markers removes the need to model crossover interference, simplifying the already extensive probability calculations. Markers that are really close together, and therefore in linkage disequilibrium (LD), break this assumption and can result in an overestimation of linkage (the end result being an inflated LOD score [44, 52, 1]), limiting the number of markers that can be considered jointly and, in turn, the resolution of the analysis.

Known recombination fractions between markers

Historically, the need for knowing the recombination fractions between markers limited the number of markers available because genetic maps (and hence, recombination fractions) need to be derived empirically. The densest genetic maps of SNPs produced to date are a 10,000 [55] and a recent $\sim$290,000 [56] marker map both provided by deCODE genetics. If these do not cover the necessary

SNPs, the HapMap project [18] helpfully provides interpolated genetic distances between an enormous number of known SNPs [31].

Correctly specified genetic model

The genetic model consisting of the penetrance function and appropriate allele frequencies for the population in question, must be correctly specified.

Known marker order

The correct ordering of all markers in the map must be known. In practical terms, this is less of an issue thanks to the Human Genome Project [64, 123], however, in different builds of the human genome, the position and ordering of markers can change.

No monozygotic twins

Monozygotic twins bias the analysis as their meioses were not independent, making their genotypes identical.

Breaking these assumptions may adversely affect the analysis, and therefore the veracity of the results.

3.2 Elston-Stewart Algorithm

Elston and Stewart provided the first general algorithm for the calculation of pedigree likelihoods [27]. The Elston-Stewart algorithm considers a pedigree to be a graph where individuals are nodes and the relationships between them edges. At its core, the algorithm performs an operation called *peeling*, so coined by Cannings *et al.* [14], to recursively collapse likelihood calculations onto pivots (an individual, the removal of which would split the pedigree in two). Peeling is used in linkage analysis to calculate the likelihood of a trait being linked to a given location on the genome.

The original algorithm could only operate on simple pedigrees. A simple pedigree is one that can be represented as a tree with a single founder couple at its root and for each nuclear family, one of the parents must also be a founder. The graph is constructed

with individuals as nodes and their relationships as edges. The tree is traversed depth first with, at a given nuclear family, likelihoods for children and the founder parent collapsed by performing a nested summation over all possible phased genotypes onto the parent connecting them to the rest of the pedigree. The complexity of this procedure is *linear* in the number of individuals in a pedigree, but *exponential* in relation to the number of markers that are to be considered jointly.

One of the first widely available software implementations, Liped [99], additionally allowed for a second founder couple. This was a prelude to the algorithm being generalised to arbitrary, acyclic graphs [14] (sometimes called 'zero-loop pedigrees') and ultimately to arbitrary graphs [15]. Further algorithmic improvements followed, e.g. Lange and Boehnke [65] showed the likelihood could be calculated by considering one individual at a time, instead of one nuclear family at a time. In addition, Lange and Boehnke showed that the sequence by which individuals are peeled affects both the run-time and the amount of memory required.

3.2.1 Scaling with Markers

The work of Botstein *et al*. [10] revealed the profusion of markers that could be derived from common genetic polymorphisms. This concept would spawn a revolution in genetic studies identifying the locations of disease causing mutations and gave impetus to further developing linkage algorithms to handle several markers simultaneously. Despite the limitations of the algorithm, throughout the 1980s, 1990s and early 2000s several Elston-Stewart derived programs emerged to extend what could feasibly be handled.

Linkage [69, 68] was one of the first programs to allow multiple loci to be considered jointly giving a true multipoint analysis. The popularity of the Linkage programs resulted in them being ported to many different systems and extended numerous times. FastLink [19] is a modified version of the original Linkage package providing faster execution times, checkpointing / crash recovery and optional parallel execution. Its improved execution time is a result of better programming, namely the use of appro-

priate caching [108] and fixed point arithmetic. Parallelism is implemented using the TreadMarks [51] system that provides a global address space across a network of workstations. Distributed shared memory architectures like TreadMarks are not commonly found on modern computer clusters, which favour message passing interfaces (e.g. MPI [37], see Section 5.2.1).

Vitesse [96] uses a genotype recoding scheme to limit the effective number of markers that are summed over and, so-called, "fuzzy inheritance" to infer the transmission probabilities of these recoded genotypes. At publication, Vitesse was able to analyse up to 8 markers jointly.

3.2.2 Discussion

Despite the utility of the Elston-Stewart algorithm and its associated family of software implementations, there remains a crucial limitation. The algorithm scales linearly with respect to the number of individuals in a pedigree, it scales exponentially with the number of markers in the multipoint case. This is not to suggest other algorithms have superseded it completely, for exceptionally large pedigrees, it remains an essential tool for providing exact LOD scores.

3.3 Lander-Green Algorithm

The use of hidden Markov models (HMM) for the purpose of genetic mapping will be limited in this survey to those algorithms derived from the Lander-Green algorithm [62]. HMMs are a general dynamic programming approach for calculating probability distributions over a set of unobserved (hidden or latent) states in a system that is assumed to be a Markov process [104]. Dynamic programming is a term used to describe any algorithm where the answers to smaller common subproblems are retained or *cached* in memory and reused at successive stages of the algorithm, avoiding recalculation.

At its core, the Lander-Green algorithm seeks to calculate the probability distribution over all possible ways segregation could occur at each locus across a chromosome

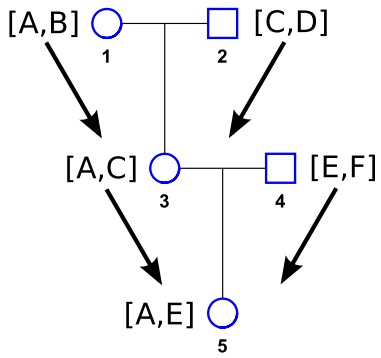


Figure 3.1: For each phased genotype, the allele on the left is maternal and the allele on the right is paternal. The pedigree in the diagram corresponds to the inheritance vector 0000. The inheritance vector is all zeros because the alleles at persons 3 and 5 are inherited from their grandmothers.

conditional on genotype information, whilst taking into account the probability distributions at neighbouring loci. Segregation through the pedigree is represented as a binary vector of segregation or meiosis indicators at each locus, termed an *inheritance vector*. For each non-founder, there are two segregation indicators, one maternal and one paternal. Each indicator states which grandparent that particular allele was inherited from (see Figure 3.1).

Assuming no genetic interference, how alleles segregate through the pedigree at each locus (corresponding to a single inheritance vector) can be modelled as a Markov chain along a chromosome, with all possible inheritance vectors forming the hidden variables of the HMM. It should be clear that this will affect scalability drastically as the number of possible inheritance vectors grows exponentially with the total number of meioses.

The original purpose of the algorithm was to infer genetic distances using simple three generation pedigrees and the Expectation-Maximisation (EM) algorithm [23]. The expectation step selecting new recombination fractions and the maximisation step employing the HMM approach to calculate the total likelihood given both map and genotype data.

The Lander-Green algorithm was adapted by Kruglyak *et al*. [57, 60] to the problem of disease mapping by calculating LOD scores using the Elston-Stewart algorithm at each locus, summing over all possible weights from the complete inheritance probabil-

ity distribution. From now on we will refer to the Lander-Green algorithm with respect to disease mapping as the Lander-Green-Kruglyak algorithm to avoid confusion with the map construction algorithm.

3.3.1 Scaling with Meioses

Over time, the problems associated with scalability have been ameliorated through extensive optimisation, many of which are exemplified by new software implementations. These can be categorised under three broad headings: state space reductions, evaluation method and compact representations.

State Space Reductions

Founder Symmetry, was the main speed-up featured in Homoz [57] and generalised in Genehunter version 1.0 [60], which identified that inheritance vectors differing only in the founder's phase are equivalent. This is quite intuitive as no information with regard to founders' parents (by definition) are included in the pedigree.

Founder Couple Reduction, was first used in Allegro version 1.0 [38]. Given that both members of a founder couple are untyped, then from the perspective of the analysis they are indistinguishable from one another and any calculations related to them need only be performed once. This might seem like a trivial reduction, however, eliminating only a single meiosis to be summed over cuts the complexity in half.

Inheritance State Space Reduction. Genehunter version 2.1 [80] describes an iterative algorithm to identify and take advantage of situations where observed genotypes invalidate certain inheritance vectors, giving them a likelihood of zero. This is not just time, but space efficient as these inheritance vectors do not need to be held in memory.

Evaluation Method

The *Idury-Elston Algorithm* [47] describes a divide and conquer algorithm to incrementally work on successive bisections of the inheritance space. Divide-and-conquer algorithms take large, perhaps otherwise intractable, problems and split them recur-

sively into smaller problems, the answers to which are combined to solve the original problem. This not only provided a speed-up compared to early versions of Genehunter, but even when it had been superseded by more efficient approaches, was employed in the Merlin [3] software package as it could be adapted to Merlin's internal hierarchical representations of inheritance data. The Idury-Elston algorithm was succeeded by the use of *Fourier transforms* [59], speeding up the standard forward-backwards algorithm approach from $O(n^2)$ complexity to $O(n \cdot log(n))$.

Compact Representations

Sparse Binary Trees, used in the Merlin software package [3], explicitly exploit symmetries in the pedigree and genotype data resulting in improved memory usage and execution time. This is achieved by the observed genotypes invalidating whole classes of inheritance vectors and by generalising the state space reduction heuristics described above, making them a formal part of the data structure employed.

Multi-Terminal Binary Decision Diagrams (MTBDD) were employed in Allegro version 2.0 [39] to hold probability distribution data. MTBDDs are graphical representations of binary functions, which make gains from exploiting uniqueness (two trees holding the same data must be the same) and non-redundancy (no two subtrees are unique) to achieve compression. Note, however, this does not speed up analysis, it just makes it easier to fit in memory in situations where there is redundancy to exploit. This is only the case when a majority of individuals are typed. Allegro 2.0 must still explicitly use the state space reduction techniques to lower the run-time.

3.3.2 Extensions

Haplotype Reconstruction

One advantage to using a hidden Markov model for linkage analysis is that the complete model can be used to perform haplotype reconstruction in the same run. The result of haplotype reconstruction is the complete specification of all phased alleles for all members of the pedigree at all markers. In this case, haplotypes are inferred from the

pedigree and the genotype data. The Viterbi algorithm [124] is used to find the most likely sequence of hidden states and the most likely assignments of founder alleles for this sequence of inheritance vectors found by enumerating all possibilities.

Special Considerations for SNPs

Special attention has recently been paid to SNPs. SNPs have special characteristics that, when taken into account, can both improve run-time and results. The density of SNPs allows for a shortcut when evaluating the HMM, as transitions from one inheritance vector to the next can be safely restricted to a small number of recombinations. This reduces the number of calculations enormously, but still provides a good approximation [3]. Secondly, the exceptionally dense SNP sets currently available provide many SNPs that are in strong LD with one another. If there are many small families in a given linkage study, then SNPs in high LD with one another can be clustered into pseudo-markers, avoiding the bias normally associated with high density markers [1].

Parallel Processing

An HMM approach, like the Lander-Green-Kruglyak algorithm, is easily applicable to parallelism [17]. This is due to the evaluation of an HMM consisting of very many independent substeps: the calculation of forward probabilities, backward probabilities and finally the integration into the complete inheritance distribution all provide opportunities to exploit parallel processing (see Section 4.3).

3.3.3 Discussion

The complexity of the Lander-Green-Kruglyak algorithm scales linearly with respect to the number of markers, but exponentially with the number of meioses, due to the need to enumerate all inheritance vectors. Linkage programs based on the Lander-Green-Kruglyak algorithm are the dominant methodology for genetic linkage analysis in humans as a majority of linkage projects involve small families (larger ones are harder to accurately collect) that use many markers to perform genome-wide analysis. Much

work has been done to extend its applicability to larger pedigrees, but with the number of calculations doubling with each meiosis (therefore quadrupling with each additional person added to the pedigree), there is still a hard limit to what is possible given the computing power and memory available. Even with compact representations, such as the MTBDD inspired Allegro 2.0, it is only capable of making large savings when a large majority of the individuals in the pedigree are typed.

3.4 Bayesian Networks

Bayesian networks allow for dependencies between variables to be made explicit and the optimal ordering of calculations identified, saving time and memory. The existence of such orderings in the Elston-Stewart algorithm were identified previously [65], but the problem was not framed as a Bayesian network until much later [29, 70]. The more general approach permits variables to be eliminated in a less strict order. Contrast this with the Elston-Stewart algorithm that only eliminates variables by working up the pedigree and the Lander-Green algorithm that does so across loci. The use of Bayesian networks enables exact likelihood calculations to scale to larger problems than previously.

SuperLink [29] employs a Bayesian network to optimise the ordering of computations in order to run on problems outside of the capabilities of FastLink, Vitesse and Genehunter. SuperLink has been superseded by SuperLink-Online [110] which runs as a loosely-coupled distributed system (specifically Condor Grid Middleware [77]) across potentially thousands of personal computers.

Both programs dynamically decide how to order calculations. In the case of large pedigrees with few markers, an Elston-Stewart-like peeling sequence results and in the small pedigree, many marker case, the ordering resembles the Lander-Green algorithm. In cases not at these extremes, a greedy heuristic is used to decide the variable elimination ordering. These procedures have been extended by using simple stochastic techniques that extend the permissible problem size considerably [30]. In addition, varia-

tional inference has been applied to reduce run-time, but the results are not as accurate as other linkage programs [32].

3.4.1 Discussion

Whilst Bayesian networks can be used to perform exact likelihood calculations on problems outside of the range of complexity the Elston-Stewart and Lander-Green algorithms are capable of handling, it is no silver bullet. Even if we knew the globally optimal ordering for variable elimination, it would not be difficult to find an input that would exhaust available time and/or memory.

The only algorithms currently capable of handling any size of pedigree and number of markers are stochastic in nature and provide approximations of the likelihoods needed for assessing linkage.

3.5 Stochastic Simulation

Whilst the process of stochastic simulation has been independently developed several times [83], it came to prominence during the Manhattan Project where Monte Carlo simulations were used to model nuclear detonations, after which it closely followed the development of stored instruction electronic computers. The seminal work of Metropolis *et al*. [84] went on to describe what we today call Markov chain Monte Carlo (MCMC) and was later generalised by both Hastings [41] and Green [36].

Stochastic simulations are used to produce approximations to otherwise intractable problems by sampling. The precursor to MCMC, the Monte Carlo method, produces samples at random from a probability distribution, but, for problems with many dependent variables, is inefficient, due to the large proportion of low likelihood or invalid states generated. By utilising a Markov chain of possible states, MCMC can go from one valid state to the next using a random walk. MCMC aims to sample a state x with probability proportional to the distribution function $\pi(x)$ (the distribution function is not a probability as we do not normalise it). MCMC tends to converge faster than the

Monte Carlo method as it will focus on the lowest energy (most likely) states which have a greater impact on the results, but, despite this, MCMC can still suffer from long convergence times. We discuss MCMC in more detail in Section 4.1.

With respect to genetic linkage analysis, whilst there was some early use of the Monte Carlo method [101], it was not until the late 1980s through to the 2000s that MCMC was investigated. The initial work during this period focused on optimising a pedigree's *descent state* via MCMC [66, 67, 112] and sequential imputation [49]. The descent state of a pedigree is a combination of both an inheritance vector (in this literature called a *descent graph*), which states *how* alleles are inherited, and the set of founder alleles, which determine *what* is inherited. MCMC-based linkage analysis was later improved by optimising over descent graphs, that have a smaller state space than descent states (as they omit the specification of founder alleles). The likelihood of a given descent graph is the summation over all possible descent states, which must be enumerated. The first application to use the descent graph representation was Simwalk2 [111, 113], which used the Metropolis-Hastings algorithm to perform MCMC.

3.5.1 Descent Graph Samplers

MCMC is a general framework to solve problems involving the simulation of a probability distribution. The difference between algorithms will come down to the details of individual samplers. A sampler is a means to generate, based on the current state of the Markov chain, the next random state to be considered. An enormous number of such samplers are available in the literature, but here we will focus on those which sample descent graphs and have had the greatest impact:

Simwalk2 sampler

The Simwalk2 sampler [111, 113] consists of a number of transition rules to tweak the current descent graph at a single locus, e.g. changing the source of one of a non-founder's alleles from paternal to maternal (or vice-versa). To improve mixing and to ensure irreducibility for markers with more than two alleles, the number of these transitions performed per step of the chain is a geometrically

distributed random variable of mean 2. When operating on many tightly linked markers, the Simwalk2 sampler performs badly due to poor mixing. In addition, with many markers run-time can become prohibitive. However, for larger pedigrees with very few markers Simwalk produces accurate results in a short time [126].

Whole locus Gibbs sampler

The locus sampler [42] is a block-Gibbs sampler which operates on a single locus and samples from all possible meiosis indicators at that locus, conditional on flanking markers. The locus sampler an extension of the Elston-Stewart algorithm that stores all intermediate likelihood calculations for use in a sampling step called reverse peeling. Unfortunately, the locus sampler can get stuck in low probability states due to the creation of highly unlikely double recombinations and, by virtue of this, mix poorly when using tightly linked markers.

Whole meiosis Gibbs sampler

The meiosis sampler [121] operates on a single meiosis indicator running through all loci using a scheme based on the Lander-Green-Kruglyak algorithm. The meiosis sampler alone may not produce an irreducible chain, but this is dependent on the actual genotype data operated on (see Section 4.3.3). The meiosis sampler mixes well with tightly linked markers, but suffers poor mixing in larger pedigrees with missing data.

LM-sampler

The LM-sampler [121] is not a new sampler in itself, but a combination of both the locus and meiosis samplers. The LM-sampler features an additional parameter which states the probability with which each sampler is selected at each step of the simulation. So long as the split between samplers is not more extreme than 1:4, it mixes well, leveraging the advantages of both samplers [126].

Multiple meiosis sampler

The multiple meiosis sampler [122] is a generalisation of the meiosis sampler,

updating several meioses at a time across all loci, to aid mixing. The sampler is evaluated using the factored hidden Markov model (FHMM) algorithm described by Fishelson and Geiger [30]. A major disadvantage of the multiple meiosis sampler is that it is far more computationally intensive than the single meiosis equivalent and can suffer poor mixing if none of the individuals of the selected meioses are typed.

Haplotype sampler

The haplotype sampler [122] is a special case of the multiple meiosis sampler that requires the user to state *a priori* an individual for which the pedigree can be effectively split into two smaller pedigrees that are amenable to exact calculation. The haplotype sampler is highly accurate, but not generally applicable to all pedigrees, for example, in an inbred pedigree there may not exist a single individual that can split the pedigree into two components.

All of these samplers, with the exception of the Simwalk2 sampler, are available in the Morgan software package [120] (derived from the Loki package [42]), which comprises a large collection of programs where each embodies one or a mix of the sampling techniques detailed above. Despite this, Simwalk2 is the mainstay of geneticists working with large pedigrees. Simwalk2 is popular because it is both well established and can handle arbitrary pedigrees.

3.5.2 Discussion

While MCMC allows us to tackle problems that are intractable to evaluate precisely they have numerous disadvantages. Firstly, long convergence times mean simulations can be slow. Given that we do not want the starting point of the simulation to directly bias the outcome, the chain must be "burnt-in" for sufficient time. This amounts to running the chain to forget the seed and is essentially wasted work. Secondly, the sequential nature of MCMC tends to restrict it to serial execution. This is less of an

issue for applications where the likelihood function takes a bulk of the time and could be written to be multithreaded. For linkage analysis it is more complicated as calculating the likelihood of an individual descent graph takes very little time, instead it will be necessary to parallelise the samplers themselves.

4 Statistical Methods For Linkage Analysis

Essentially, all models are wrong, but some are useful.

George E. P. Box

Previously in chapter 3 we touched upon all the major methods to evaluate the likelihoods required for multipoint genetic linkage analysis that have been used to date. In this chapter we will detail the samplers and algorithms necessary to stochastically simulate inheritance in pedigrees that are too large and complex to be tackled by deterministic, exact methods.

We start with a review of the concept of Markov chain Monte Carlo and proceed by describing the descent graph structure used to specify inheritance. After describing the two main block Gibbs samplers our work is based on, we finish by describing the procedure used to obtain LOD scores conditional on marker phenotypes.

4.1 Markov Chain Monte Carlo

Markov chain Monte Carlo (MCMC) is a method for randomly sampling a high dimensional space for which the enumeration of all states is too costly and for which direct sampling is otherwise difficult.

MCMC works by generating samples linked by a Markov chain. The Markov chain is constructed such that the chain spends more time in the most important regions, so

that the samples approximate the target distribution, $\pi(x)$. $\pi(x)$ cannot be sampled directly, but can be evaluated up to a normalising constant. Each step in the chain is selected randomly from all of the potential next steps that could be made from the current state, specified by some proposal mechanism, $q(x' \mid x)$. The sequence of steps form a random walk through the state space. In order to simulate the target distribution, not all proposed steps are taken. Given the current state in the Markov chain x and the proposed next step x', the acceptance probability, $\alpha(x, x')$ in the case of the Metropolis–Hastings algorithm, is:

$$\alpha(x, x') = min\left(1, \frac{\pi(x')\, q(x \mid x')}{\pi(x)\, q(x' \mid x)}\right)$$

The next step in the Markov chain is set to x' with probability $\alpha(x, x')$ or else it stays as x, resulting in a transition probability of:

$$p(x' \mid x) = q(x' \mid x)\, \alpha(x, x')$$

The proposal mechanism is crucial to MCMC working correctly. If the chain gets stuck in one part of the state space (for example, because new proposals are always rejected), then the chain is said to *mix poorly*. Poorly mixing chains produce biased simulations as they do not sample the target distribution properly.

For any starting point, the chain will converge to the target distribution so long as it is both *irreducible* and *aperiodic*. An irreducible chain is one where, for any state in the Markov chain, there is a positive probability of visiting all other states. An aperiodic chain is one that does not get trapped in cycles. A Markov chain will be both irreducible and aperiodic if we ensure that the *detailed balance* condition is satisfied:

$$\pi(x')\, p(x \mid x') = \pi(x)\, p(x' \mid x)$$

As we cannot guarantee that the starting position of the Markov chain is from the target distribution (if it was, it would imply that we can sample from it directly), we need to

burn-in the chain by discarding all samples for a sufficient time. MCMC can suffer from long convergence times due to the amount of burn-in required for some problems. Convergence times are greatly reduced by performing an initial optimisation step to start the chain with a high likelihood initial state (we investigate several methods for optimisation of the initial state in Section 4.5).

4.1.1 Gibbs Sampling

Gibbs sampling [34] is another method of MCMC and a special case of the Metropolis–Hastings algorithm. The main difference comes from the insight that where exact algorithms exist, we can use these to directly sample full conditional distributions. A full conditional distribution of $\pi(x)$ is obtained by keeping all components of x constant except one, x_i, and sampling as a function of x_i alone. We will refer to all components in x apart from x_i as x_{-i} The proposal distribution is defined as:

$$q(x' \mid x, x_{-i}) = \pi(x' \mid x_{-i})$$

Substituting the proposal distribution above into the Metropolis–Hastings acceptance probability will always output a value of 1 and is therefore always accepted:

$$\begin{aligned}
\alpha(x, x') &= min\left(1, \frac{\pi(x' \mid x_{-i})\, q(x \mid x', x_{-i})}{\pi(x \mid x_{-i})\, q(x' \mid x, x_{-i})}\right) \\
&= min\left(1, \frac{\pi(x' \mid x_{-i})\, \pi(x \mid x_{-i})}{\pi(x \mid x_{-i})\, \pi(x' \mid x_{-i})}\right) \\
&= 1
\end{aligned}$$

This does not contradict the statements made previously as the chain can still mix poorly because the proposed sample, x', may be equal to x, which would be analogous to a rejection in the Metropolis–Hastings algorithm.

Before we can go into the details of the Gibbs samplers used throughout this thesis, we must first describe the *descent graph* which will serve as the state defining each step of the Markov chain.

4.2 Descent Graphs

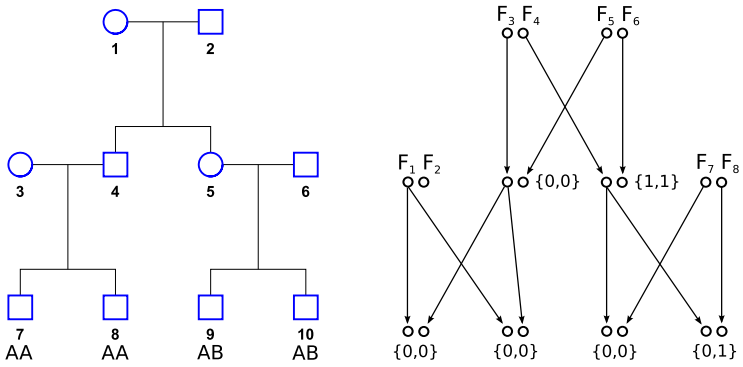
Genetic descent graphs specify the paths by which founder alleles are inherited by each member of a pedigree [111, 58]. Much of the early work on applying stochastic techniques to linkage analysis [66, 67, 112] utilised genetic descent states, which are a combination of a descent graph coupled with concrete assignments for each founder allele (namely, not just *how* alleles are inherited, but *what* alleles are inherited as well). Descent graphs are preferred over descent states as the state space is smaller and they avoid some of the irreducibility problems associated with descent states [109].

Descent graphs specify gene flow information by means of *meiosis indicators*. Each non-founder in a pedigree has a set of two meiosis indicators, one from each parent. The maternal meiosis indicator, for example, will have a value of 0 or 1 to indicate that, at this particular locus, genetic material was inherited from the maternal grandmother or grandfather, respectively.

A descent graph is stored in memory as a binary vector where individuals' meiosis indicators are found in an arbitrary, but defined, order. For a more concrete example see Figure 4.1a, which contains a simple three generation pedigree where only the last generation is typed. There are six non-founders (individuals 4, 5, 7, 8, 9 and 10) each of which have two inbound arrows in the more informative descent graph representation. Figure 4.1b represents which parental allele was inherited. Assuming the ordering of individuals specified previously, this descent graph would correspond to the binary vector `[00 11 00 00 00 01]`. By convention, the maternal meiosis indicator is stated first, followed by the paternal meiosis indicator for each individual in turn.

So far we have only used a single locus as an example, but this generalises to all loci typed in a pedigree. Suppose we have n individuals of whom f are founders, each single locus descent graph contains $2(n-f)$ meiosis indicators and therefore in order to specify all m markers a descent graph of size $2(n-f)m$ is required.

As we do not specify the actual assignment for founder alleles, we need to be able to enumerate all possible founder allele assignments for a given descent graph and cal-



(a) Simple pedigree (b) Descent graph

Figure 4.1: Simple pedigree in (a) composed of three generations where only the last generation is typed and (b) shows one possible descent graph with founder alleles labelled F_1–F_8. If the values of all founder alleles were specified, then this would be an example of a descent state.

culate their likelihood. We will describe such an algorithm in Section 4.3.1.

For the remainder of this chapter, we will refer to the current descent graph at any step of the Markov chain as S. A particular meiosis indicator, for example the i^{th} meiosis at the l^{th} locus, will be specified as $S_{i,l}$. The wild card "$*$" will be employed to indicate all meioses at locus l as $S_{*,l}$ and a specific meiosis across all loci as $S_{i,*}$.

4.3 Whole Meiosis Gibbs Sampler

First proposed by Thompson and Heath [121], the whole meiosis Gibbs sampler is based on the Lander-Green algorithm [62]. Both use the forward-backward algorithm to evaluate a hidden Markov model. Whilst the Lander-Green algorithm efficiently enumerates all combinations of meiosis indicators at all loci, the exponential complexity prevents scaling to large numbers of meiosis. The whole meiosis sampler instead samples realisations of $S_{i,*}$.

For the meiosis sampler, we keep all but the meiosis being sampled fixed and sample that meiosis across all loci. We are aiming to compute:

$$P(S_{i,*} \mid \{S_{k,*}, k \neq i\}, \mathbf{Y})$$

The forward component of the algorithm involves calculating the cumulative proba-

bility for the meiosis indicator $S_{i,l}$ given loci up to and including locus l. The first locus does not depend on any previous loci:

$$Q_1(S_{i,1}) \propto P(Y_1 \mid S_{*,1})$$

All proceeding loci are calculated with:

$$Q_l(S_{i,l}) \propto P(Y_l \mid S_{*,l})P(S_{i,l} \mid S_{i,l-1})$$

The probability of recombination, $P(S_{i,l} \mid S_{i,l-1})$, is defined as follows:

$$P(S_{i,l} \mid S_{i,l-1}) = (Q_{l-1}(S_{i,l-1})(1-\theta_{l-1}) + Q_{l-1}(1-S_{i,l-1})\theta_{l-1})$$

where θ_l is the recombination fraction between locus l and $l+1$.

To understand how to calculate the likelihood of the genotype data given the descent graph, $P(Y \mid S)$, we must first understand about founder allele graphs as this probability can then be calculated by enumerating all legal combinations of founder alleles given the current descent graph and marker phenotypes.

4.3.1 Founder Allele Graphs

To calculate $P(Y \mid S)$ we must enumerate all possible assignments to each founder allele at a given locus conditional on both marker data and the current descent graph. Whilst we could literally enumerate all possibilities, a more efficient approach is to bound the search by iteratively constructing a *founder allele graph*.

At each locus we can construct a founder allele graph that is independent of all other loci, making the assumption of linkage equilibrium. A founder allele graph is composed of two nodes per founder in a pedigree, one per allele. Nodes are connected by undirected edges where the descent graph passes through a typed individual and that edge is labelled with the individual's genotype at that locus.

These edge labels are conditions that must be satisfied. For example, if the nodes for

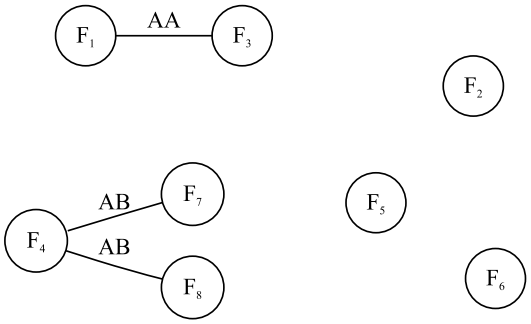


Figure 4.2: Founder allele graph corresponding to Figures 4.1a and 4.1b. Founder allele nodes F_1 and F_3 are connected by genotype edge AA, implying there is only one legal founder allele assignment for that connected component: A = A and C = A.

founder alleles X and Y are connected by an edge labelled homozygously AA, then in all enumerations of founder allele assignments, the founder alleles X and Y must both be A. Founder allele graphs are not necessarily connected, but instead are composed of one or more connected components. It can be seen from the edge constraints that each component has either one or two legal founder allele assignments, based on whether any of the edges are labelled with a homozygous genotype or not.

The founder allele graph for our previous example in Figures 4.1a and 4.1b is given in Figure 4.2. Nodes F_2, F_5 and F_6 are singletons, because those founder alleles did not pass through any typed individuals based on the descent graph. There is only one valid assignment for the component $\{F_1, F_3\}$, where both are A's. The other component $\{F_4, F_7, F_8\}$ has two possible assignments: (A, B, B) and (B, A, A).

Now that we have a set of components, C, for each of which we have defined one or two allele assignments, a, we can finally calculate the likelihood $P(Y \mid S)$ by:

$$P(Y \mid S) = \prod_{i=1}^{n} P(C_i)$$

where

$$P(C_i) = \sum_{i} \prod_{j} P(a_{i,j})$$

and for any component composed of a single founder allele node, with no edges connecting it to other nodes or itself, then $P(C_i) = 1$.

4.3.2 Sampling

Once we have calculated the"forward" cumulative probability, the last term can be normalised to get the distribution of $S_{i,L}$ given marker phenotypes and meiosis indicators at all preceding loci. $S_{i,L}$ is sampled from:

$$Q_L(s) = P(S_{i,L} = s \mid \{S_{k*}, k \neq i\}, \mathbf{Y} = Y^{(L)})$$

All other meiosis indicators for $S_{i,*}$ are sampled iteratively from the right-most locus $L-1$ to locus 1. This forms the "backwards" phase of the forward-backward algorithm. If we have already sampled meiosis indicators at loci $j = l, \ldots, L$, then all other meioses are sampled from:

$$P(S_{i,L} = s \mid \{S_{k,*}, k \neq i\}, \{S_{i,j}, j = l, \ldots, L\}, \mathbf{Y}) = \frac{Q_{l-1}(s)P(S_{l,i} \mid S_{l+1,i})}{Q_{l-1}(0)P(S_{l,i} = 0 \mid S_{l-1,i}) + Q_{l-1}(1)P(S_{l,i} = 1 \mid S_{l-1,i})}$$

4.3.3 Irreducibility Issues

The whole meiosis sampler is known to mix well in cases where there are many tightly-linked loci. Unfortunately, this comes at a price; dependent on the data, the whole meiosis sampler alone is not guaranteed to produce an irreducible Markov chain.

The canonical example of this, credited to Lin and Speed [76], is given in Figure 4.3. Neither child has any alleles in common, any valid descent graph requires more than one meiosis indicator to be changed to get to another valid descent graph.

4.4 Whole Locus Gibbs Sampler

To ameliorate the shortcomings of the whole meiosis sampler, while retaining its advantages, it is common practice to combine it with another sampler that is guaranteed to produce a valid Markov chain [121, 74]. Fortunately, the whole locus Gibbs sampler, whilst known to mix slowly with tightly-linked markers, is guaranteed to be irreducible

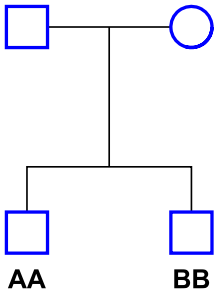


Figure 4.3: Any valid descent graph describing this pedigree and marker data will require more than one meiosis indicator to change in order to reach another valid descent graph, thus breaking the irreducibility assumption.

given all recombination fractions are positive.

The locus sampler was envisioned by Kong [54], but first implemented by Heath [42] to perform QTL mapping. In much the same way as the meiosis sampler is an adaptation of the Lander-Green algorithm, the locus sampler owes much to the Elston-Stewart algorithm and its various improvements. The locus sampler proceeds in a forward phase, that we refer to as *peeling* to ascertain the likelihood of the pedigree and its associated marker phenotype data and a backwards phase (*reverse peeling*), where we sample from partial likelihoods to produce a realisation of $S_{*,l}$.

4.4.1 Pedigree Peeling

The Elston-Stewart algorithm is a method to recursively collapse the likelihood calculations of a pedigree given marker phenotype data onto *pivot* individuals at the intersections of nuclear families. This procedure is termed *peeling*.

A pivot is a node that, when removed, splits a connected graph component into two new components. In Figure 4.4, individual 3 is the pivot separating nuclear families A and B. Given that all the pivot's descendants can *only* inherit material from the pivot's ancestors via the pivot, it makes sense that all descendant trait probabilities need only be described in terms of the transmission probabilities from the pivot.

Using this intuition, we consider a pedigree in terms of its constituent nuclear families and peel these away one at a time until we have exhausted the pedigree. Once we know the sequence of nuclear families to be peeled, we consider each in turn by

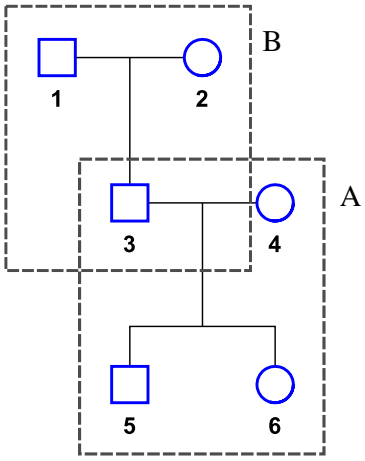


Figure 4.4: A simple three generation family, split into nuclear families A and B. The pivot of nuclear family A is individual 3.

applying different *peeling operations* on individuals within the family.

To peel a simple pedigree, like the example in Figure 4.4, we must define two operations; $PeelChild$ and $PeelPartner$. With these two operations we can process all individuals in the order 6, 5, 4, 3, 2, 1. In the following examples we will refer to this peeling sequence as P and the i^{th} individual in the sequence as P_i.

Peeling children

The goal of $PeelChild$ is to peel a child node given its marker phenotype and replace it with a partial likelihood in terms of the parent genotypes. Analogous to the concept of the pivot, both parents form the *cut-set* of the child. We will define the cut-set of the pivot, p, as the set of unpeeled individuals that lie on the threshold of the peeled subset of the pedigree after p itself has been peeled. With the pedigree in Figure 4.4, if individual 6 has the genotype **AB**, we need to work out the likelihood of this given all possible combinations of parental genotypes.

Note that the genotype **AB** is *unphased*, i.e. there is no information to describe where either allele came from. When performing these likelihood calculations, it is simpler to deal with *phased* genotypes. The unphased genotype **AB** can correspond to either of the phased genotypes AB (A from the mother, B from the father) and BA (vice-versa).

We need to calculate two probabilities: the *penetrance probability*, $P(Y_i \mid g_i)$, and the *transmission probability*, $P(g_i \mid g_m, g_p)$. The penetrance is a function of whether the phased genotype g_i is legal given the unphased genotype and therefore has the value 0.0 or 1.0. If individual 6 is typed with the unphased genotype **AB**, the penetrance function is:

$$P(Y_i \mid g_i) = \begin{pmatrix} P(Y_i \mid AA) \\ P(Y_i \mid AB) \\ P(Y_i \mid BA) \\ P(Y_i \mid BB) \end{pmatrix} \quad P(\mathbf{AB} \mid g_i) = \begin{pmatrix} 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \end{pmatrix}$$

For the transmission probability function, we need to calculate the probability of inheriting each phased genotype from the parents. To do this, we enumerate all possibilities for the two parental genotypes and normalise the resulting matrix. It is not important that some of these will clearly be negated by either parents' actual genotypes because we have not yet got to them in the peeling sequence. Hence:

$$P(g_i \mid g_m, g_p) = \begin{pmatrix} P(g_i \mid AA, AA) & P(g_i \mid AB, AA) & P(g_i \mid BA, AA) & P(g_i \mid BB, AA) \\ P(g_i \mid AA, AB) & P(g_i \mid AB, AB) & P(g_i \mid BA, AB) & P(g_i \mid BB, AB) \\ P(g_i \mid AA, BA) & P(g_i \mid AB, BA) & P(g_i \mid BA, BA) & P(g_i \mid BB, BA) \\ P(g_i \mid AA, BB) & P(g_i \mid AB, BB) & P(g_i \mid BA, BB) & P(g_i \mid BB, BB) \end{pmatrix}$$

$$P(AB \mid g_m, g_p) = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.125 & 0.0625 & 0.0625 & 0.0 \\ 0.125 & 0.0625 & 0.0625 & 0.0 \\ 0.25 & 0.125 & 0.125 & 0.0 \end{pmatrix}$$

This is trivial to compute because the transmission of each allele is independent. So the probability $P(g_i \mid g_m, g_p) = P(g_{i_m} \mid g_m)P(g_{i_p} \mid g_p)$ (where g_{i_m} is the maternal allele for the i^{th} individual and g_{i_p} the paternal allele). The probability of transmitting each allele can take the values 0.0, 0.5 or 1.0, if g_{i_x} is the same as zero, one or both of the parental alleles, respectively.

We can now peel away the child by replacing it with a probability function in terms of its parental genotypes:

$$PeelChild(P_i) = \sum_{g_i} P(Y_i \mid g_i)P(g_i \mid g_m, g_p)PreviousOp(P_{i-1})$$

$PreviousOp$ is defined as whatever the previous operation was on the preceding person in the peeling sequence, which will, by definition, be a function in terms of the genotypes of P_i or, if they are independent, in the case of siblings, just sum together with the current function. In the case where there was no previous operation, i.e.: if this is the first person in the peeling sequence, then the probability is 1.

Peeling partners

Assuming we have already peeled both individuals 5 and 6 using the $PeelChild$ operation, next we need to peel individual 4, the partner of the pivot to nuclear family A. The cut-set of this operation will just be the pivot individual. The operation $PeelPartner$ is similar to $PeelChild$, but because founders lack any parents, instead of a transmission probability, we need to use the *founder probability*, $P(g_i)$. The founder probability is the probability that such a genotype would be found in the population as a whole, which we know because it is specified by the markers minor allele frequency. Therefore, given a minor allele frequency, f_i, the founder probability is defined as:

$$P(g_i) = \begin{pmatrix} P(AA) \\ P(AB) \\ P(BA) \\ P(BB) \end{pmatrix} = \begin{pmatrix} (1-f_i)^2 \\ (1-f_i)f_i \\ f_i(1-f_i) \\ f_i^2 \end{pmatrix}$$

Finally, we must include the results from our $PeelChild$ operations indicated again as $PreviousOp$.

$$PeelPartner(P_i) = \sum_{g_i} P(Y_i \mid g_i)P(g_i)PreviousOp(P_{i-1})$$

We can continue applying the $PeelChild$ and $PeelPartner$ operations iteratively until we only have one individual left. The likelihood function that remains is $P(g_I \mid Y_l)$.

4.4.2 Peeling Conditional on Flanking Loci

So far, our peeling operations have only been dependent on the marker data at the current locus. For the locus sampler, we want to sample meiosis indicators conditionally on the flanking loci. In order to achieve this, we must replace the single locus transmission probability, $P(g_{c,l} \mid g_{m,l}, g_{p,l})$, with:

$$P(g_{c,l}, S_{c_m,l-1}, S_{c_m,l+1}, S_{c_p,l-1}, S_{c_p,l+1} \mid g_{m,l}, g_{p,l}) =$$
$$P(g_{c,l} \mid g_{m,l})P(S_{c_m,l-1} \mid g_{c_m,l}, g_{m,l})P(S_{c_m,l+1} \mid g_{c_m,l}, g_{m,l})$$
$$P(g_{c,l} \mid g_{p,l})P(S_{c_p,l-1} \mid g_{c_p,l}, g_{p,l})P(S_{c_p,l+1} \mid g_{c_p,l}, g_{p,l})$$

Note that we now indicate which locus every meiosis indicator and genotype comes from with the subscript l, to remove ambiguity, whereas previously everything had been at a single locus. The new transmission probability adds four terms, but as they are all similar, we will only expand one. As before, θ_l is the recombination fraction between loci l and $l+1$:

$$P(S_{c_m,l-1} \mid g_{c_m,l}, g_{m,l}) = \begin{cases} \theta_{l-1}^{S_{l-1,c_m}}(1-\theta_{l-1})^{1-S_{l-1,c_m}} & \text{if } g_{c_m,l} = g_{m_m,l} \neq g_{m_p,l} \\ (1-\theta_{l-1})^{S_{l-1,c_m}}\theta_{l-1}^{1-S_{l-1,c_m}} & \text{if } g_{c_m,l} = g_{m_p,l} \neq g_{m_m,l} \\ \frac{1}{2} & \text{if } g_{c_m,l} = g_{m_m,l} = g_{m_p,l} \end{cases}$$

Otherwise, the peeling calculations are identical.

4.4.3 Sampling

The process of sampling in the whole locus sampler takes place in two discrete steps. First, we backtrack through the pedigree in the reverse order sampling phased genotypes conditioning on the genotypes already sampled in a process known as *reverse peeling*.

Second, we must convert these sampled phased genotypes into meiosis indicators to update the descent graph.

Reverse Peeling

At the final individual in the peeling sequence, P_I, we obtain $P(g_I, S_{l-1}, S_{l+1}, Y_l)$, from which phased genotypes for P_I can be sampled by:

$$P(g_I \mid S_{l-1}, S_{l+1}, Y_l) = \frac{P(g_I, S_{l-1}, S_{l+1}, Y_l)}{\sum_{g_{I,i}} P(g_{I,i}, S_{l-1}, S_{l+1}, Y_l)}$$

To sample from a partner (the equivalent of the $PeelPartner$ operation) we sample from:

$$P(g_i \mid g_{i+1}, Y_1, \ldots, Y_i) \propto P(Y_{des(i)}, S_{des(i),l-1}, S_{des(i),l+1} \mid g_i, g_{i+1}) P(g_i, Y_i)$$

And for each child from:

$$P(g_i, S_{c_m,l-1}, S_{c_m,l+1}, S_{c_p,l-1}, S_{c_p,l+1} \mid g_m, g_p, Y_1, \ldots, Y_i) \propto$$
$$P(Y_i \mid g_i) P(g_i, S_{c_m,l-1}, S_{c_m,l+1}, S_{c_p,l-1}, S_{c_p,l+1} \mid g_m, g_p) P(Y_{des(i)}, S_{des(i),l-1}, S_{des(i),l+1} \mid g_i)$$

In the above formulae, we use the notation $des(i)$ to indicate all the descendants of individual i, though in reality these are just the ones that precede i in the peeling sequence.

As one backtracks through the peeling sequence, the current individual's genotypes will be defined by a partial likelihood function conditional on the genotypes of individuals that had not been peeled yet. As we have sampled all of these genotypes during the reverse peel, they now have definite values. Using what we have sampled up to this point, the partial function will collapse to a function conditional only on the current individual's genotypes. At this point it is normalised and sampled.

The complete process of reverse peeling will obtain a realisation of phased genotypes for all individuals at locus l. In order to be complete, however, we need to convert these phased genotypes into a realisation of $S_{*,l}$.

Sampling Meiosis Indicators

To convert phased genotypes into meiosis indicators, we need to consider whether each meiosis is informative. An informative meiosis is one where the allele was inherited from a heterozygous parent, as the genotype contains phase information it will dictate whether the parent's maternal or paternal allele was inherited.

If the parental genotype is homozygous, then it is uninformative and the meiosis indicator must be sampled conditionally on meiosis indicators at flanking loci:

$$P(S_{l,c_m} = 0 \mid S_{l-1,c_m}, S_{l+1,c_m}) \propto \theta_{l-1}^{1-S_{l-1,c_m}}(1-\theta_{l-1})^{S_{l-1,c_m}}\theta_l^{1-S_{l+1,c_m}}(1-\theta_l)^{S_{l+1,c_m}}$$

Once all meiosis indicators have been set, we have a new realisation of $S_{*,l}$.

4.4.4 Generalising to Arbitrary Pedigrees

So far, we have only described how the locus sampler works on simple pedigrees. In order for peeling to be generally applicable, we need to describe two further extensions to the peeling operations, i.e. how to peel parents down to a child pivot and how to handle inbred pedigrees. Neither of these will affect the basic methodology of reverse peeling and sampling.

Peeling parents

It can be seen that if we limit ourselves to just the $PeelChild$ and $PeelPartner$ operations, we will be unable to process an arbitrary outbred pedigree. For example, if the parents of individual 4 in Figure 4.4 were specified, then $PeelPartner$ would not be appropriate as it does not take into account the transmission probability of genotypes between individual 4 and her parents. To enable peeling of arbitrary outbred pedigrees, we need to define the operation $PeelParent$.

$PeelParent$ is essentially identical to $PeelPartner$, however, it includes the transmission probability for the child pivot it is peeling down to. This makes intuitive sense if you think that the peeling of individuals from a pedigree is like removing them from

it entirely. If the parents were removed, then we would not apply the transmission probabilities for the child because it would be a founder.

$$PeelParent(P_i) = \sum_{g_i} P(Y_i \mid g_i)P(g_i)P(g_c \mid g_i, g_j)$$

Note that the transmission probability is conditional on the genotypes of the individual being peeled (g_i) and their spouse (g_j). The spouse is peeled next using the $PartnerPeel$ operation as usual.

$PreviousOp$ has not been mentioned. This is because the inclusion of downwards peeling has created a problem. Not only can $PreviousOp$ be from the ancestors or the descendants of the individual being peeled, but we can create peeling sequences where there are two previous operations, one from below and one from above. Peeling becomes even more complicated if we want to peel arbitrary graphs like inbred pedigrees. This is because in a graph containing a cycle, the definition of what nodes are above and below is not clearly defined.

Inbred pedigrees

In a simple outbred pedigree, we can clearly define all of a pivot's ancestors (nodes above it in the graph) and all descendants (nodes below it in the graph) as two distinct sets. In the case of an inbred pedigree, the likelihood will be affected by multiple gene flows down the pedigree, implying that we need to peel off individuals conditional on members of the pedigree outside of their immediate family.

In Figure 4.5, if individual 9 has already been peeled as usual with $PeelChild$, who is next in the peeling sequence? If we peel individuals 7 or 8 using another $PeelChild$ in the usual manner, then we will not include the gene flow from the other side of the pedigree. We could peel individual 8, for example, but the cut-set of the operation would be $\{5, 6, 7\}$, not just $\{5, 6\}$ as would normally be the case. The addition of individual 7 to the cut-set will not actually affect the mechanism of $PeelChild$ in any way, but it will add a dimension to the probability function. When we later peel individual 7, we

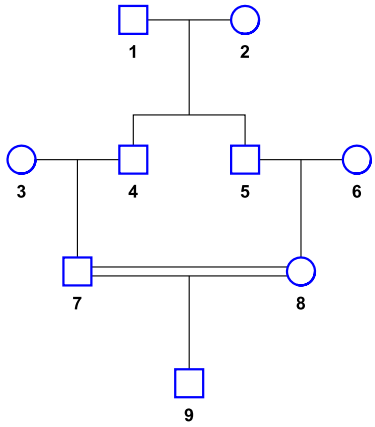


Figure 4.5: An inbred pedigree showing a first-cousin marriage.

have already begun calculating the likelihood of its genotypes in terms of the other side of the family and this will be accessed in the usual way through $PreviousOp$. When the peeling sequence reaches the other side of the inbreeding loop, the extra dimensions will cancel themselves out and we will have taken into account all possible gene flows.

In the case of an outbred pedigree, peeling off one nuclear family at a time is guaranteed to produce an optimal peeling sequence, in the sense that the dimensionality of each probability function is minimised. In the case of an inbred pedigree, this is not the case and we need to perform an optimisation procedure to select the best peeling sequence to minimise the amount of work we have to do. The amount of work we have to do is directly proportional to the size of the cut-sets during peeling operations.

We will delay describing methods for finding suitable peeling sequences until Section 6.4. A complete rundown of the many algorithms to do this can be found in the review by Thomas [119].

4.5 Starting States

The basic operation of a Markov chain requires that the previous state have been a legal descent graph. Given that, it is necessary to provide a bootstrapping mechanism to provide an initial configuration from which we can take our first step in the chain. In addition to this, we should state that the starting point of the chain can be crucial to the convergence time. Were we to start the chain with a low probability state, it will take

a long time to reach equilibrium, so we want to start with a higher probability state. However, as we do not know anything *a priori* about the shape of the state space, we might benefit from a lower probability starting state in, for example, a situation where there are local maximum that are hard for the Markov chain to escape. This will not improve performance, but it will at least inform us of the existence of poor mixing.

4.5.1 Single Locus Sampling

We can use the existing whole locus Gibbs sampler to generate realisations of the descent graph at each locus conditional on marker phenotypes at that locus alone and ignore the flanking markers. This will not produce the best starting state, but it will provide a suitably random one if we want to explore other parts of the space starting from points of lower probability. We will use this as a primitive to perform a more advanced technique called *sequential imputation*.

4.5.2 Sequential Imputation

In sequential imputation, we randomly select a locus, l, and run the single locus sampling algorithm described above. We then iteratively work from locus $l + 1$ to locus L using the regular locus sampler, but we sample with respect to the flanking locus on the left *only*. Next, we work from locus $l - 1$ to locus 1 sampling with respect to the flanking locus on the right only.

This will provide us with a valid descent graph to use as a starting state, however we might have been unlucky and not produced the best we could have obtained. So, we run the entire process n times (for some suitable value of n) and select the state with the highest probability as the starting state of the Markov chain.

4.6 LOD scores

To generate LOD scores, we take the samples generated by the Markov chain and calculate the likelihood of a disease trait at positions of interest along the chromosome. The

disease trait is a marker like any other, the difference being that whereas we know the locations of SNPs, we are using likelihood calculations to estimate where the disease trait is located. This likelihood is calculated using the standard peeling algorithm we described in Section 4.4.1.

4.6.1 Sobel-Lange Estimator

Previously, we have used Y to denote the marker phenotypes, which up to now have only consisted of the genotypes of typed individuals. A pedigree also defines which individuals are affected, unaffected and of unknown affection. We will label the affection status of an individual as the trait phenotype Y_T in contrast to marker phenotypes, now Y_M.

The Sobel-Lange estimator [67] calculates the LOD score at the current position being considered, over the set of all sampled descent graphs $\hat{S}$. It is calculated as follows:

$$P(Y_T \mid Y_M) = \sum_{\hat{S}} P(Y_T \mid \hat{S}) P(\hat{S} \mid Y_M) \propto \frac{1}{n} \sum_{i=1}^{n} P(Y_T \mid \hat{S}_i)$$

The final LOD score at a given location is calculated by standardising and taken to log_{10} as convention dictates:

$$LOD = log_{10} \left(\frac{P(Y_T \mid Y_M)}{P(Y_T)} \right)$$

where $P(Y_T)$ is the likelihood of the trait ignoring the neighbouring loci in the descent graph.

4.7 Summary

In this chapter, we described the necessary background to all the statistical techniques employed at different stages of this thesis.

We began by describing descent graphs, an abstract representation of gene flow

through a pedigree and two methods to simulate this flow via a Markov chain. We described the operation of the whole meiosis Gibbs sampler that samples a single meiosis across all loci in a chromosome creating a realisation of $S_{i,*}$ and the whole locus Gibbs sampler that uses pedigree peeling at a single locus to produce a realisation of $S_{*,l}$. We closed by describing procedures similar to the whole locus sampler that can be used to generate legal starting configurations for the Markov chain as well as produce LOD scores at arbitrary locations.

The details about the relative performance of each sampler was touched upon; the meiosis sampler mixes well with tightly linked markers, but suffers from potential irreducibility issues, whereas the locus sampler mixes slowly, but guarantees irreducibility. This issue will be investigated in more detail in the next chapters related to the implementation and evaluation of these samplers and their parallel counterparts.

5 Parallel Sampler Design

> The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it.
>
> *Steve Jobs*

In this chapter, we want to outline the design of SwiftLink and how our parallel Gibbs samplers for genetic linkage analysis will work. First, we need to look at the hardware and software techniques available to us and specifically take a closer look at how massively parallel graphics card architectures will influence our work. We conclude by looking at the main system components (locus sampler, meiosis sampler and LOD score code) in turn and sketch how parallel versions of each will function.

5.1 Computing Hardware

In search of ever increasing speed, computer processor manufacturers have turned to ideas from parallel processing and distributed systems research. This move has forced a paradigm shift in how software is designed and implemented. We will briefly outline why exploiting parallelism has become a necessity.

5.1.1 Multicore Processors

Despite the exponential increase in processor speeds between 1965 (when Gordon Moore predicted a doubling of transistor count every two years) and 2005 [117], single

core processors have been replaced by *multicore* processors. This was due to a large number of factors, not least of which being physical limits, such as heat dissipation, and the complexity of current single core processors.

Multicore processors feature multiple independent processors fabricated into a single integrated chip. Each core will have its own level 1 (L1) cache, but will generally share a common L2 or L3 cache with other cores in the same processor. Unlike previous advances in single core processors, such as improved instruction-level parallelism or branch prediction, existing programs are not immediately capable of taking advantage of the extra processing power. In this way multiple processor cores are not transparent to the programmer because they must be taken advantage of *explicitly*. Whilst this may have been unfamiliar ground for programmers, hardware manufacturers had been using parallel hardware for non-general purpose processing for years. The main example being hardware accelerated computer graphics.

5.1.2 Graphics Cards

Computer graphics co-processors have existed since the beginning of the personal computing era. Graphics cards usually come in the form of a separate daughter board comprising discrete processing capabilities and memory dedicated to the task of accelerating both 2D and 3D transformations. Increasingly, they are found in integrated settings as well, on motherboards and CPUs. Throughout, we will use the terms *graphics card* and *GPU* (graphics processing unit) interchangeably, to mean a piece of hardware, the main purpose of which, is to perform graphics related functions. Strictly speaking, however, a GPU is to a graphics card, what a CPU is to a computer.

Graphics cards came to prominence in the mid-1990s as real-time 3D graphics became a major selling point of consumer PC hardware. One of the first true 3D games, *Quake* by ID software [46], is widely credited as being instrumental in their adoption [71, 127]. Whilst no software could rely on every PC having 3D graphics acceleration until the early 2000s, standards emerged to ease their use for developers. These included SGI's (now Khronos group's) *OpenGL* [98] and Microsoft's *DirectX* [85].

A graphics card will be faster than performing the same operation on a standard CPU because computer graphics operations are relatively trivial to parallelise and GPUs are *massively* parallel. GPUs would originally implement these operations directly in hardware, but increasing demands saw the introduction of programmable shaders to both OpenGL and DirectX in the early 2000s. Shaders are small programs which perform additional processing to textures and geometry, e.g. adding motion blur in a post-processing stage to a racing game. As these capabilities became more complex with the addition of floating point arithmetic and control flow structures, GPUs began to converge with more general purpose CPU architectures. How GPUs are programmed has recently become standardised by several groups in the form of Nvidia's *CUDA* (Compute Unified Device Architecture) [94], Khronos group's *OpenCL* [97] and Microsoft's *DirectCompute* [86]. A graphics card that can perform arbitrary computation is called a *GPGPU* (general purpose graphics processing unit).

5.1.3 Manycore Processors

The current trend of increasing numbers of CPU cores will continue with the push to more *heterogeneous* architectures, CPU/GPGPU combinations and different cores on a single chip like the IBM Cell architecture [45]. Whilst GPGPU programming might be temporary, it will probably be due to a convergence of the two architectures, i.e. multicore leading to manycore. In this sense we want to use current GPGPU hardware to start to address the kinds of scaling concerns that will exist in a world of hundreds of general purpose CPU cores.

5.2 Parallel Programming

Parallel processing on general purpose CPUs is a broad topic, so we will limit ourselves to a discussion of the most common methods in use today. In addition, we will limit ourselves to those aspects that require explicit utilisation by the programmer.

5.2.1 Task Parallelism

Task parallelism is a *coarse-grained* parallelism where different blocks of computation are independent and thus can be carried out simultaneously on the same or different computers, with or without cooperation. We will focus on the main methods of task parallelism commonly used.

Processes

The unit of execution at the level of the operating system (OS) is the *process*. A process is an instantiation of a running program in memory. By default it executes instructions sequentially in a single *thread* of execution and is sandboxed from other processes by running in its own *virtual address space*. In the case where the programmer wants to perform multiple actions at the same time, another process can be *forked* and they can be programmed to cooperate on the same task. Unfortunately, this is not very convenient as the virtual address space of one process is inaccessible to the other process [1] and it is cumbersome to manually craft messages to be sent between processes, for example, via pipes or sockets. Instead, we can spawn a new *thread* within the same process and have both threads working cooperatively with access to the same address space.

Threads

Threads do not need to communicate explicitly to work cooperatively, but can do so implicitly by interacting with shared data structures in memory. This works well in the read-only case, but otherwise may lead to lost or corrupt data as a result of conflicting writes. This problem is overcome by ensuring data structures are *thread-safe* by forcing threads to contend for access. There are numerous ways to force contention among threads: locks, mutexes, semaphores, signals, monitors, etc. All these methods are in general quite hard to get right and incorrect use will often lead to problems such as *race conditions* (where a program bug is intermittent because it only appears in certain

[1] Obviously, there are even ways around this limitation through shared memory, but we will ignore these details to simplify the discussion.

timing scenarios) and *deadlocks* (lock dependencies resulting in threads unintentionally preventing one another from running). The complexity of proper thread-based programming has lead many to the conclusion that it is simply the wrong model of concurrency to pursue [73].

Message Passing

Message passing avoids the problems associated with multithreaded applications by forcing communication via objects called *messages* that are delivered to recipients via queues. The recipient drains these messages asynchronously decoupling the communication mechanism between sender and recipient.

In the multi-process example described earlier, we mentioned using a socket to pass packets of data between processes. One advantage of this approach is that the socket can be identified by another process on another host on the network. In this *distributed* case, threads are unsuitable as each process does not even exist in the same *physical address space*. MPI (Message Passing Interface) [37] is a library that implements this concept and simplifies inter-host communication specifically for this purpose. MPI is found on most computer clusters. In MPI, the messages are encoded in network packets and the senders and recipients of these messages are processes that can sit on different computers in a cluster.

Message passing generalises to arbitrary software entities running on the same or different computers. The *Actor model* of concurrency, implemented by the Scala [107] and Erlang [28] programming languages, treat software systems as inherently concurrent entities that communicate with one another via message passing. Instead of messages being passed between physical hosts, they are passed between virtual entities on the same computer.

5.2.2 Data Parallelism

When algorithms require the same calculations to be performed on multiple inputs these are examples of data parallelism, which by their very nature are more *fine-grained* than

the units of work found in task parallelism.

SIMD

Processors that support SIMD (single instruction, multiple data) can take a single instruction, for example, the addition of a constant to the data at a memory location, and apply it to multiple data elements that are specified contiguously in memory. Code employing SIMD is said to be *vectorised*. Whilst many vector architectures have existed over the years, the most commonly used ones are Intel MMX [48], which was superseded by the current SSE instructions in the x86 architecture, and the AltiVec [25] instructions in IBM's PowerPC architecture.

SIMD instructions can deliver an enormous boost to performance for applications that can be vectorised, the "killer-application" being multimedia. Unfortunately, not all code can be vectorised and its application requires quite arcane knowledge of assembler-level instructions. Several compilers, including recent versions of GCC, have begun to support auto-vectorisation of code [6], but the resulting binaries are unlikely to achieve the gains of hand-coding.

SIMT

SIMT (single instruction, multiple threads) is a term coined by Nvidia to describe CUDA. CUDA is similar to an SIMD architecture, in that each instruction is run on multiple data items held in contiguous memory, and a multithreading architecture, where everything can be written in a high-level programming language. Whereas SIMD instructions can often be added at a later date or be optional extras selected at compile-time, applications for GPGPU tend to require complete rewrites of entire modules in order to express the problem in terms of the CUDA programming model.

5.3 Graphics Card Programming

We will explore both the GPGPU hardware architecture and how this maps to programming abstractions defined by the CUDA specification to promote high performance.

CUDA was chosen as it is said to have a shallower learning curve compared to OpenCL, however, OpenCL operates in much the same way as CUDA.

5.3.1 Architecture

The GPU architecture has many similarities with a modern CPU but there are some important differences. Figure 5.1 shows single core CPU and GPU architectures in diagrammatic form. Both have access to a large quantity of RAM, often of the same order of magnitude. Whereas the CPU dedicates most chip space to different layers of cache in order to perform more intelligent sequential execution and opportunistically avoid the longer round-trip times to RAM, the GPU is specialised towards compute intensive processing. The outcome of specialisation is not only more space devoted to ALU (arithmetic logic unit), but less control and cache.

Figure 5.1: Diagrammatic representation of single core CPU architecture and GPU. Whereas CPUs have layers of cache and optimised execution, GPUs favour increased compute capacity. (image credit: Nvidia [20])

Each GPU features numerous *multiprocessors* (shown as rows in Figure 5.1), where each multiprocessor has many *CUDA cores*. A CUDA core is different from a CPU core as it does not have a dedicated fetch-decode unit to process incoming instructions, instead, each multiprocessor has a single fetch-decode unit forcing all CUDA cores to operate in lockstep on the same instruction in parallel. To facilitate programming such a large hierarchy of compute resources, CUDA defines its own programming model. This is different from just providing a library of functions to be used from the CPU, but dictates a strict manner to structure code. The programming model maps down precisely with the hardware and presents opportunities for scaling beyond the device tested on.

5.3.2 Programming Model

CUDA is programmed in CUDA C, an extension of the C programming language, which has recently started to support much of the C++ syntax [93]. CUDA C adds a number of keywords that are used by applications programmers to explicitly use the features of the underlying hardware.

A piece of code that runs on a GPU is called a *kernel*. Kernels are analogous to C functions, but, whereas the invocation of a C function will operate on its parameters within a single thread of execution, a CUDA kernel will implicitly create many light-weight threads, the number and dimensionality of which is explicitly controlled by the programmer. The presence of a host PC is central to the CUDA model of execution. This host / device split permits a far more rigid programming model on the GPU than would otherwise have been possible as we will see in the following sub-sections.

CUDA kernels are called synchronously (by default) from the host PC. In order for any non-trivial kernel to function properly, memory must be dynamically allocated in device memory from the host and any data copied from host to device memory. In order to retrieve the results from any computation, a similar copy must be performed from device to host memory (see Figure 5.2 for an overview). These copies must be limited as much as possible as they are comparatively very slow operations.

Hierarchy of Thread Groups

The CUDA execution model is based on a hierarchy of thread groups. Each thread belongs to a *block* and each block is the member of a *grid*. Sub-problems are mapped to blocks in a task parallel fashion. Each block contains a set of threads to complete the data parallel actions necessary to complete that sub-problem. Each thread is similar to a CPU thread, however, they are expected to work in lock-step executing the same instructions on different variables. Whilst thread execution paths can diverge, for example, due to an *if* statement, this will slow down execution substantially as all threads that do not take that path will *stall*.

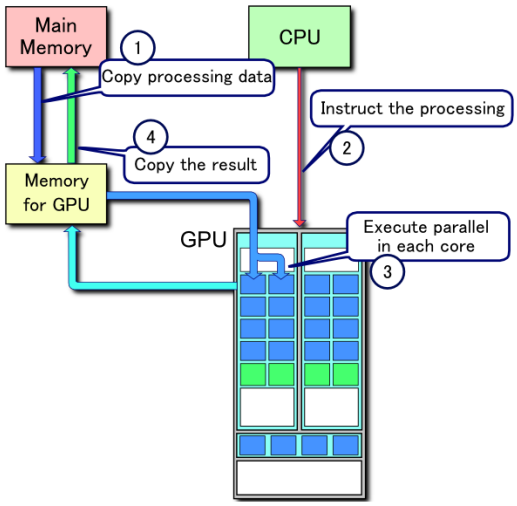


Figure 5.2: Flow of execution between host PC and CUDA device (image credit: WikiMedia Commons [21])

How different blocks and threads map to different elements of problems is left up to the programmer, who is provided with two predefined variables by the CUDA environment: **blockIdx** and **threadIdx**, that are used as offsets to index data being worked on. Each block is run by the CUDA scheduler on a separate multiprocessor to permit threads within the same block to communicate, which they do through *shared memory*. Each thread is run by a single CUDA core in the multiprocessor assigned to that particular block. Grids describe the set of blocks that are working on similar sub-problems and are only important if the GPU is executing multiple kernels concurrently.

Shared Memory

Threads within the same block can communicate with one another through the use of shared memory. Shared memory is a software-defined cache where the programmer decides what is held for quick retrieval. Access times for shared memory are much lower than the graphics card's main memory.

Acknowledging the complexity in multithreaded code on CPUs (see section 5.2.1), CUDA does not have any locking mechanisms to prevent contention for shared resources. Instead, it is expected that memory accesses be carefully designed around using a different thread per memory offset so that no two threads are accessing the

same data. The one concurrency primitive that is provided is *barrier synchronisation*.

Barrier Synchronisation

As the programmer defines the number of threads to be run for each block, the hardware must define a smaller quantum of threads that it will actually run in parallel. This quantum is called a *warp* and all CUDA-enabled GPUs define it as a constant of 32 threads. This is important because the hardware will only run each warp of 32 threads in lock-step, not all those in a block. That means that different warps require an additional mechanism to synchronise their execution paths within the same block to permit the serialisation of different phases of parallel events. CUDA provides a primitive called barrier synchronisation which operates by forcing all threads within the same block to halt execution until all threads have reached this point. A barrier is defined using a call to the function **syncthreads**.

5.3.3 Compute Capability

To ensure backwards compatibility and provide a simple way to target a complete generation of CUDA GPUs, each graphics card will support a clearly defined sub-set of features. This is referred to as the device's *compute capability* and each CUDA program must state clearly which version compute capability it has been written for. Therefore, if a graphics card has compute capability n, it cannot run software that assumes compute capability $n + 1$. For a complete list of all CUDA compute capabilities to date, the reader is referred to the extensive list in appendix F of the Nvidia CUDA programming guide [93].

5.4 Approach

The goal of this thesis is to appropriately map the algorithmic elements of the different Gibbs samplers detailed in the previous chapter with the hardware and software platforms described. The key challenge will be balancing the level of hardware utilisation

while trying to retain the same level of accuracy as a single-threaded implementation. Our basic approach for designing a parallel implementation of an existing sampler is to try to retain the serial ordering of events as much as possible. We should only break this rule if there is no other choice or it permits a far higher degree of parallelism and we are able to show it does not impact the accuracy of the result.

The two main platforms we will be considering are multicore CPUs and current generation GPGPUs. Any software focused on multicore CPUs should be capable of being parallelised to the same number of threads as there are cores, or, in the case of Intel CPUs, twice the number of cores if hyperthreading is enabled. GPUs require us to identify substantial data parallelism as well as task parallelism in order to be effective. The selection of hardware was essentially based on what is commonly available in an average PC but is currently underutilised. We feel that these choices are sensible with a view to the future as we will likely be using similar, though increasingly parallel, hardware for many years to come. We will state where different ideas are intended for multicore CPUs or GPUs. Here, we only state aspects of coarse-grain task and fine-grained data parallelism that can be exploited to accelerate each individual sampler. We do not investigate any parallelism schemes related to the Markov chain mechanism itself, such as, parallel tempering or other multiple chain schemes that would parallelise trivially.

Parallel tempering [118] is a method of MCMC that runs multiple Markov chains at different temperatures. The temperature of the chain affects the acceptance probability, i.e. the higher the temperature, the greater the acceptance probability. Pairs of chains are coupled by performing a separate metropolis update on whether they should exchange states. Parallel tempering increases the level of exploration as the "hotter" chains can sample more randomly and pass on higher likelihood states to the "colder" chains. For more details see the review article by Earl and Deem [22]. A parallel implementation of parallel tempering could use a single thread per Markov chain. These threads would only need to communicate to exchange states from their chains.

5.4.1 Libraries

For multicore CPU, we will use threads as they remain the dominant model for parallel processing under UNIX. For many applications, threads map directly to specific tasks, for example, a video game may have a rendering thread, an A.I. thread and a networking thread. This could be achieved using, for example, the pthreads library, however, for scientific computing there is a simpler alternative in the form of OpenMP. OpenMP works by the programmer annotating different parts of the program's control flow with compiler directives telling the compiler what work needs to be farmed off to worker threads in a thread pool. This permits the same code to be used for single and multithreaded CPU implementations as calls to the OpenMP library can set the number of threads in the thread pool dynamically. OpenMP has been supported natively in the GNU compiler collection (GCC) since version 4.2, so is widely available.

For random number generation on the CPU we use the GNU scientific library and its implementation of the Mersenne Twister [81] algorithm, which is a popular random number generator for Monte Carlo simulations and has nice C++ bindings. For GPGPU programming, we use CUDA 4.0. For parallel random number generation, we use the built-in cuRAND library provided by Nvidia.

5.4.2 Input files and parameters

To be backwards compatible with other linkage analysis software and data formatting utilities, we will support the "linkage"-style input files. A user will be required to supply a pedigree file, a map file and a data file containing the necessary input data for the project (for further details see appendix A.1.4).

The main interface will be command-line based and the user will be required to specify all the necessary parameters to run the Markov chain including: the number of iterations to be discarded as burn-in, the number of iterations of simulation, how frequently samples from the chain are to be scored, the probability of running the locus sampler, how many threads should be spawned and whether the GPU should be used

at all (see appendix A.1.2 for a complete list). We give some insight into how these parameters should be set in chapter 7, however it should be noted that, for any given project, the optimal settings will be data dependent.

5.4.3 LOD scores

LOD score calculations offer extensive opportunities for both task and data parallelism. With regard to task parallelism, all LOD score calculations are independent as the calculation of $P(Y_T \mid \hat{S})$ is dependent only on the descent graph, S, which is fixed. Therefore, if we have m markers and want to calculate a LOD score at t equidistant points between each marker, then calculating the LOD scores for a single descent graph can be parallelised in $t(m-1)$ ways. This will be more than sufficient to saturate multicore CPUs for years to come.

Data parallelism is trickier but possible if we parallelise each individual peel operation. Each peel operation involves the construction of an n dimensional matrix, where n is the length of the cut-set for that peel operation (see Section 4.4.1 for more details). For example, peeling a child up to two parents would have a cut-set of size 2 (both parents) and involve 16 independent likelihood calculations. The real benefit of this scheme will come from highly inbred pedigrees as these would require larger cut-sets at multiple points in the peeling sequence.

One question is how to actually apply this to the GPU. Should peeling the entire pedigree be a CUDA kernel? Perhaps not, as the number of threads required is not constant throughout the peeling sequence. In that case, should we have a kernel per peeling operation to ensure the number of threads is the same as the size of the matrix? It is unclear what exactly the granularity of sub-problem assigned to each CUDA block should be; this might result in a high degree of overhead as the host would need to make several kernel calls per LOD score. These issues will be investigated empirically in the next chapter related to implementation.

5.4.4 Locus Sampler

The locus sampler can utilise exactly the same technique for data parallelism that we described for LOD scores. It will be additionally followed by a sequential phase performing the sampling of phased genotypes and meiosis indicators.

Task parallelism has to be handled differently. Each task in question will still be the peeling of the whole locus, but now they are no longer independent. The locus sampler creates a realisation of $S_{*,l}$ by conditioning on marker phenotypes at locus l and meiosis indicators at flanking loci. Therefore, when the sampler is run at locus l, the meiosis indicators at loci $l-1$ and $l+1$ must remain fixed. At the extreme, this suggests we could run the locus sampler in two batches: first all the even numbered loci and then all the odd numbered loci. Unfortunately, the single-threaded locus sampler performs a scan across all loci, not in sequential order, but in a pseudo-random order. Thus, we should try to retain this process as much as possible as it will limit any second order effects from the ordering of variable updates. With this in mind, we will define a window size of w loci and permute a list of offsets into this window. For example, if the window size was 3 and we permuted the list of offsets from $(0, 1, 2)$ to $(1, 2, 0)$, then we would run three batches of the locus sampler, each batch being loci $\{i, w + i, 2w + i, 3w + i, \dots\}$, where i is the current offset. The effect of the value of w will be investigated in the next chapter.

5.4.5 Meiosis Sampler

For the meiosis sampler, we cannot run multiple instances concurrently as we can with the locus sampler because the sampled realisations of $S_{i,*}$ will interfere with one another's calculations of $P(Y \mid S)$. However, all calculations of $P(Y \mid S)$ are independent of one another under the assumption of linkage equilibrium, so can be run in parallel. For the forward-backwards algorithm, we have no choice but to run it single-threaded, but at least it is of limited complexity if $P(Y \mid S)$ has already been calculated.

The meiosis sampler will be problematic to implement for the GPU. Firstly, there

is no real data parallelism, only the task parallelism we have identified so far. Even if there was, there is large thread divergence as with many graph-based algorithms. What we will do instead is take the approach mentioned in [43], which involves using one warp of threads per locus forcing the GPU to behave like a multicore CPU.

We will investigate how well the forward-backward algorithm will perform on the GPU. If it does not perform very well we might investigate a window approach similar to the one used for the locus sampler, i.e. only sampling based on a shorter window of loci.

5.5 Summary

The goal of this chapter was to investigate the design possibilities open to us to implement parallel Gibbs samplers for linkage analysis. We began by describing the different hardware we have available to us and highlighted that current GPGPU architectures resemble how CPU processors might look in the future as they trend towards manycore architectures. We then summarised the main parallel programming techniques available to us on any UNIX-like operating system and discussed both coarse-grained task parallelism and fine-grained data parallelism. CUDA programming was looked at in some detail as it differs in small, but significant ways from CPU programming that will not only influence design, but necessitate a complete rewrite of even code intended for a multicore processor. We then finished by outlining the design for our parallel samplers and LOD scoring code based on properties of the analysis and observations of the platforms we intend to target. In the next chapter, we will detail the complete system and justify the values for any undefined parameters that might be necessary for parallelism.

6 Software Implementation

> It is sort of depressing when it becomes clear that it is more effective to do crappy parallel work than good sequential work.
>
> *John Carmack*

In the previous chapter, we outlined the designs for parallel versions of both locus and meiosis samplers as well as LOD scoring code. In addition, we highlighted specific questions that would need to be investigated in order to create both multicore CPU and the manycore GPU linkage analysis software. In this chapter, we will present a series of benchmarks informing us as to the design decisions required for different hardware platforms. We start by introducing these platforms and the datasets used for testing and proceed to systematically look at all of the aspects involved in adapting these algorithms to parallel hardware. By the end of the chapter, we will have a comprehensive overview of how the software implementation of SwiftLink performs, how the platforms differ reflected in our benchmark results and an understanding of both platforms' and implementations' relative merits.

6.1 Test Hardware

All of the benchmarks reported in this chapter were performed on the same computer running Ubuntu "Lucid" 10.04.4 LTS edition with Linux kernel 2.6.32-39 for 64-bit. The computer has an AMD Phenom II CPU with four processor cores, each clocked at 3.2 GHz and 4GB of RAM clocked at 1600 MHz. The main graphics card we used for

Property	Nvidia GTX 580
GPU	
Multiprocessors (MP)	16
CUDA cores	512 (32/MP)
Clock rate (MHz)	795
Memory	
Memory (MB)	1536
Type	GDDR5
Clock rate (MHz)	2004
Bandwidth (GB/s)	192.4
Bus Width (bits)	384

Table 6.1: Table showing graphics card specification. Note that these values differ from the Nvidia reference build as it was an "overclocked" edition.

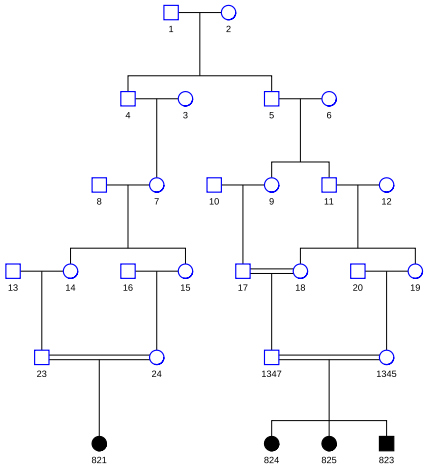
testing is an Nvidia GTX 580, manufactured by Gigabyte. Table 6.1 states all relevant specifications.

6.2 Test Pedigrees

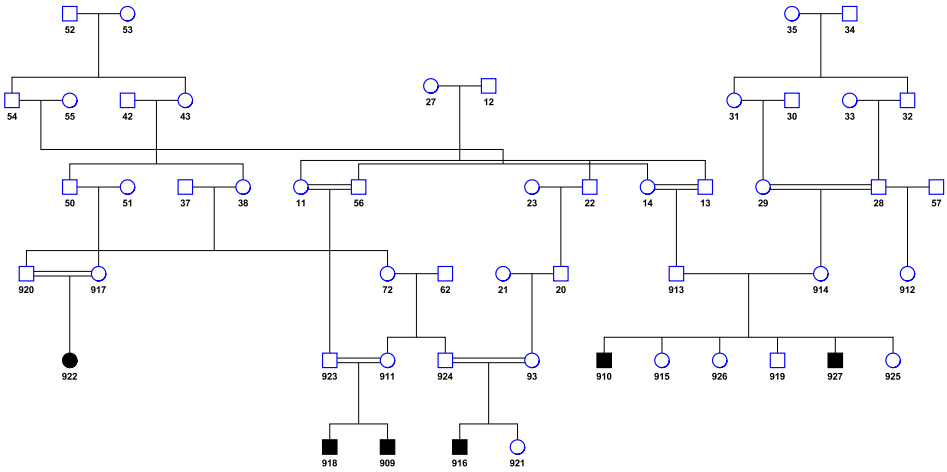
Throughout this chapter, we used two main pedigrees to test and tune different parameters. These will be referred to as the EAST syndrome and benign chorea pedigrees, which are described fully in chapter 7. In this chapter we will only use a subset of the data available for these pedigrees, namely, just chromosome 1. Both are large, multiply inbred pedigrees, typed with SNPs. The EAST syndrome pedigree (Figure 6.1a) is 26-bits, featuring 28 individuals of which 6 are typed. The benign chorea pedigree (Figure 6.1b) is 52-bits, featuring 51 individuals of which 19 are typed. These pedigrees are a true test of both the speedups and mixing of the software tested and are representative of the kind of pedigree SwiftLink was designed to handle, namely large consanguineous pedigrees.

6.3 Benchmarking Methods

Before we start analysing different aspects of our implementation, we need to explain some of the details regarding how these measurements were made.



(a) EAST syndrome



(b) Benign chorea

Figure 6.1: Inbred 26-bit pedigree, EAST syndrome, in sub-figure (a) from Bockenhauer et al. [8] and highly inbred 52-bit pedigree, benign chorea, in sub-figure (b), from Poveda et al. [102].

6.3.1 Timing

Timing measurements that are short, for example, in the millisecond range, are tricky to make accurately on a regular CPU. The ISO C99 standard states there is no guarantee of making an accurate measurement to any timing resolution (section 7.23.1 paragraph 4), which means that any method we use will be unportable. Fortunately, the x86 architecture provides such a method, as does the CUDA API.

CPU Timing with RDTSC

On the x86 architecture, the RDTSC instruction allows us to read the value of the time stamp counter (TSC) used by the processor. On old single core CPUs, this was the most accurate timing measure available. Multicore processors are more complicated as each core has its own TSC and are not guaranteed to be in sync with one another. If a process were migrated from one processor core to another between two readings being taken, then the timing measurement may be inaccurate. Further complications come from processors being capable of dynamically scaling their frequency and suspending execution altogether, so precisely what one tick of the TSC means can change between readings. To get around these problems we use two techniques. We use the POSIX standard function **clock_gettime** requesting the CLOCK_MONOTONIC clock. The only aspect it does not correct for is the effect of NTP [1] adjustments which we eliminate by turning NTP off.

Secondly, we implemented a special benchmarking mode that repeats the action we are timing 1,000 times and calculates the average. This ensures we are timing something in the range of seconds and dampen any poor measurements that might occur from ambient CPU load.

GPU Timing with CUDA Events

CUDA provides a method to asynchronously record events during application lifetime and later query when those events were completed. Events are created using the **cudaEventCreate** function, recordings made with the **cudaEventRecord** function for later inspection. The utility function **cudaEventElapsedTime** returns a float of the time between two events in milliseconds. CUDA events have a resolution of 0.5 microseconds.

[1] Network Time Protocol (NTP) automatically synchronises a computer clock despite variable network latency.

6.3.2 MCMC Diagnostics

For the most part, our parallel implementations of Gibbs samplers aim to retain the relative ordering of events from the single-threaded implementations. Where that is not possible, because doing so was too slow, we need to be able to understand the impact parallelism has on the quality of the simulation.

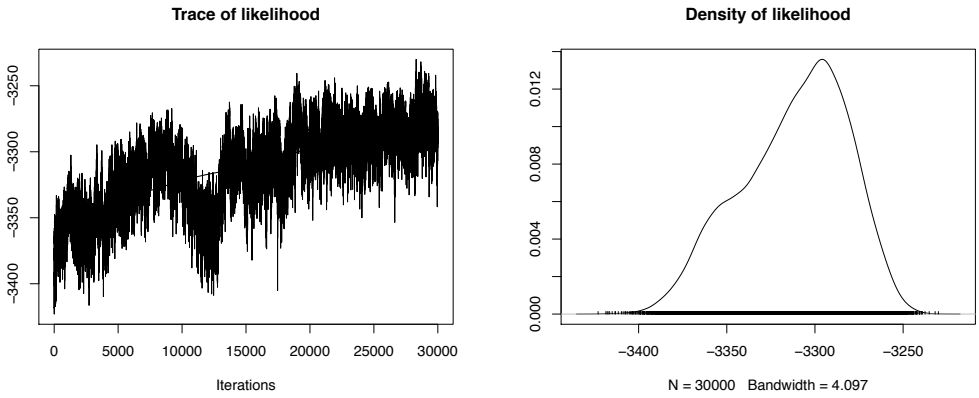
We use several MCMC diagnostics for analysis. In this section we will give an overview which is expanded on in appendix A.2. All diagnostics were performed using the R implementation of the CODA package [103]. The downside of MCMC diagnostics is that they can only help diagnose the non-convergence of a Markov chain, there is no test to state that convergence has definitely been achieved.

For each of the diagnostics, below, we ran the locus sampler, the meiosis sampler and the combined locus and meiosis sampler (where in each iteration we selected which sampler to run uniformly at random). For each run, we ran chromosome 1 of the EAST syndrome pedigree, with 780 SNPs. Each run of the Markov chain was 30,000 iterations and we did not perform any thinning of the chain.

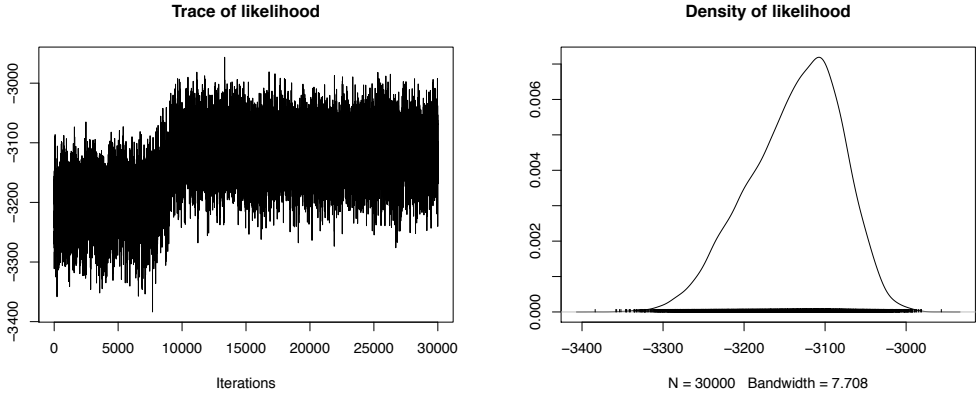
Trace and Density Plots

Trace and density plots are time series and histograms, respectively, of the likelihoods of samples from the Markov chain. We use these to see if, in running the chain, there are any obvious places where it gets "stuck" and also to compare two chains from different starting states to see if they sample parts of the space with the same range of likelihoods. Of course, this method can only show us where a chain did not converge compared to another run of the chain. If none of the chains sampled correctly, then the test will be inconclusive.

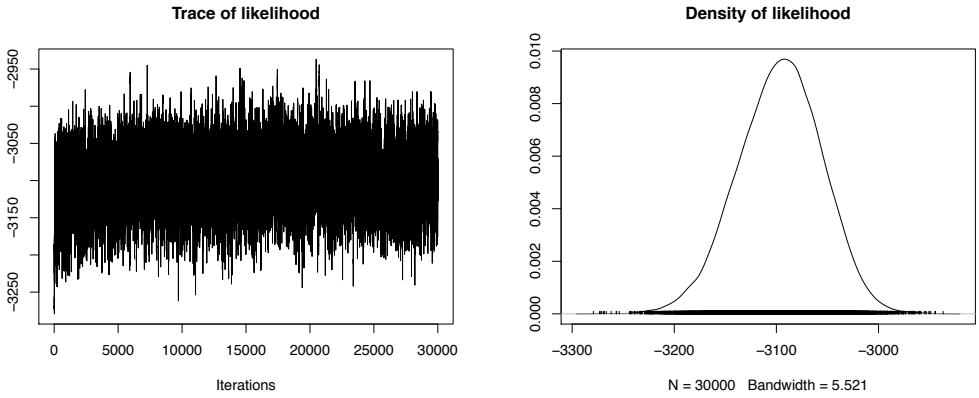
We plotted the likelihoods generated by different samplers running on the EAST syndrome pedigree. Figure 6.2a shows the slow mixing of the locus sampler. Figure 6.2b shows the meiosis sampler appearing to have converged before shifting to a higher likelihood at around iteration 10,000, perhaps due to the documented irreducibility issues, and finally, Figure 6.2c of the hybrid sampler running the locus or meiosis



(a) Locus Sampler



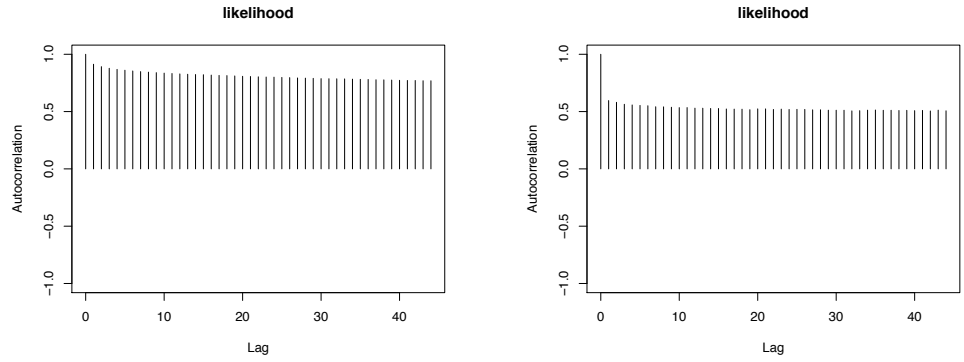
(b) Meiosis Sampler



(c) Locus and Meiosis Sampler

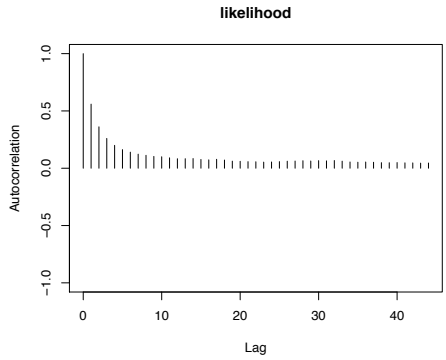
Figure 6.2: Trace and density plots of different samplers running on the EAST syndrome pedigree. Whilst the locus sampler in (a) and the meiosis sampler in (b) both appear to suffer mixing problems, using a combination of the two in (c) does not.

sampler with equal probability appears far more consistent, and appears to have converged. Multiple runs of the chain sampled the same range of likelihoods (trace plots



(a) Locus Sampler

(b) Meiosis Sampler



(c) Locus and Meiosis Sampler

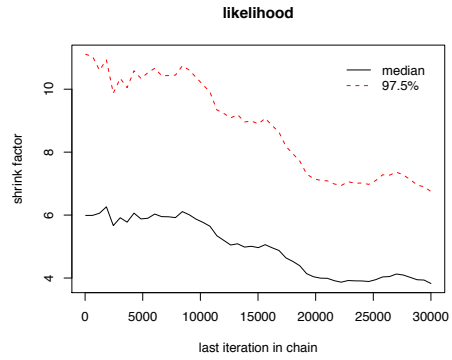
Figure 6.3: Autocorrelation plots of different samplers running on the EAST syndrome pedigree. Using both the locus and meiosis samplers together show a random spatial pattern, unlike individually.

not shown).

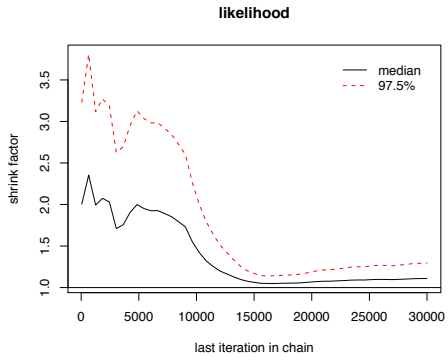
Autocorrelation

Convergence can be assessed by looking at the level of *autocorrelation* between samples from the Markov chain. The lag k autocorrelation is the correlation between every sample and its k^{th} lag. We would expect the level of autocorrelation to become smaller as the value of k increases, if it does not, then samples are correlated and indicate slow mixing of the chain.

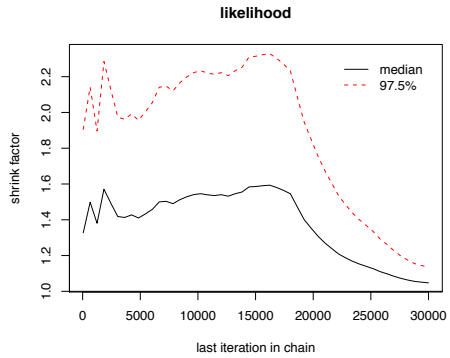
Autocorrelation statistics for the different samplers on the EAST syndrome pedigree help us to understand the results of the trace and density plots. We plotted autocorrelation up to a lag of 50 without any thinning of the chain. Figures 6.3a and 6.3b of the



(a) Locus Sampler



(b) Meiosis Sampler



(c) Locus and Meiosis Sampler

Figure 6.4: Gelman–Rubin plots of different samplers running on the EAST syndrome pedigree.

locus sampler and meiosis sampler, respectively, show significant correlation compared to the combined locus and meiosis sampler, even up to a lag of 50 (Figure 6.3c).

Gelman–Rubin Diagnostic

The Gelman–Rubin diagnostic [33] involves running the Markov chain several times from different starting positions. The chains should all behave in much the same way, so the variance *within* each chain should be the same as the variance in *different* chains. This is measured by the *shrink factor*, where, if the value is 1, it indicates that the variance within and between the chains is equal. As a rule of thumb, a value of lower than 1.05 is an indication of convergence.

Figure 6.4c shows that the combined locus and meiosis samplers only appear to converge half the time with a chain of length of 30,000 iterations, the median shrink

factor being 1.05. Rerunning with 100,000 iterations did not change the result (figure not shown). Figure 6.4a of the locus sampler appears to show that it did not converge. Figure 6.4b shows the meiosis sampler does not quite converge as often as the hybrid sampler (median shrink factor 1.11), but we know from the previous autocorrelation and trace plots that there can be times when it has problems mixing, possibly due to irreducibility. Another negative sign is that the graph is actually diverging slightly from iteration 15,000. For each of these experiments we calculated the Gelman–Rubin diagnostic across four separate runs of the Markov chain.

6.4 Finding a Peeling Sequence

Finding an optimal peeling sequence is important in inbred pedigrees in order to limit the amount of work necessary to calculate the pedigree likelihood. A good peeling sequence will directly impact the run-time of both single and multithreaded versions of the locus sampler and LOD scoring code. Below, we will describe two algorithms to find a peeling sequence: a greedy algorithm and random downhill search, evaluating the results of both on our test pedigrees.

Both optimisation algorithms require us to define a cost to each potential peeling sequence. We define the cost of a peeling sequence as:

$$CalculateCost(P) = \sum_i 4^{len(c_i)}$$

where P is the peeling sequence, c_i is the cut-set of the i^{th} individual in P, the function $len(c_i)$ returns the size of the cut-set, c_i.

6.4.1 Greedy Algorithm

We present a simple greedy approach to constructing a peeling sequence in algorithm 1. It proceeds in an iterative manner, always adding the individual from $\mathcal{U}$ (unprocessed individuals) with the lowest cost to the current peeling sequence, $\mathcal{P}$. This greedy approach is non-deterministic because of the randomness involved when multiple indi-

```
begin
    P ← {}
    U ← pedigree
    while U ≠ {} do
        for i ∈ U do
            costs[i] ← CalculateCost(P ∪ {i})
        end
        mincost ← Min(costs)
        minset ← {}
        for i ∈ U do
            if costs[i] = mincost then
                minset ← minset ∪ {i}
            end
        end
        P ← P ∪ Random(minset)
        U ← pedigree \ P
    end
end
```

Algorithm 1: Greedy algorithm to find a peeling sequence given a pedigree. Each iteration adds the individual with the lowest cost cut-set to the peeling sequence, where there is more than one, a random individual is selected.

viduals have the same cost at the current iteration. Greedy algorithms are simple, fast and tend to give good enough results for simpler problems, but as the complexity of the problem increases the range of answers increases.

6.4.2 Random Downhill Search

Random downhill search is an optimisation procedure that starts from a random point in the state space, i.e. a complete peeling sequence, and makes iterative improvements. The algorithm runs for n iterations. In each iteration, we swap a randomly selected pair of individuals in the peeling sequence and assess the cost of this new sequence. If this is an improvement (a lower cost), then we keep the change, otherwise the change is reversed. It is described more formally in algorithm 2.

```
begin
    P ← random_sequence(pedigree)
    current_cost ← CalculateCost(P)

    for i ← 1 to n do
        x ← random(len(P))
        y ← random(len(P))

        swap(P, x, y)

        tmp_cost ← CalculateCost(P)

        if tmp_cost < current_cost then
            current_cost ← tmp_cost
        else
            swap(P, x, y)
        end
    end
end
```

Algorithm 2: Random downhill optimisation of peeling sequences. The peeling sequence is permuted by swapping randomly selected indices in the sequence and the new peeling sequence accepted if it is of a lower cost.

6.4.3 Evaluation

Greedy Algorithm

We ran the greedy algorithm 20,000 times on both EAST syndrome and benign chorea pedigrees. The results are summarised in tables 6.2 and 6.3. For the EAST syndrome pedigree, whilst almost a quarter of the runs produced the lowest cost, the range was quite high and $\sim 50\%$ of the time produced a peeling sequence 10% more costly.

For the benign chorea pedigree, despite producing the optimal peeling sequence of cost 2145 $\sim 3.5\%$ of the time, the range was completely unacceptable, producing sequences greater than one and a half times that, $\sim 10\%$ of the time. The greater complexity of the benign chorea pedigree has uncovered the weakness of the greedy algorithm, which is that for problems of high complexity, it struggles to find a good solution.

Cost	Freq
625	4461
673	5553
685	2754
733	5637
745	151
781	787
793	519
841	138

Table 6.2: Results of 20,000 runs of the greedy algorithm on the EAST syndrome pedigree.

Cost	Freq	Cost	Freq
2145	699	2445	550
2193	2453	2481	328
2205	240	2493	914
2241	1621	2541	994
2253	1128	3105	235
2289	684	3153	909
2301	2058	3165	68
2337	319	3201	564
2349	2166	3213	386
2385	1187	3249	232
2397	92	3261	686
2433	706	3309	781

Table 6.3: Results of 20,000 runs of the greedy algorithm on the benign chorea pedigree.

Random Downhill Search

The random downhill algorithm was run for 1,000,000 iterations for both EAST syndrome and benign chorea pedigrees. The number of iterations was chosen arbitrarily. A single run of 1,000,000 iterations took 27 seconds for the EAST syndrome pedigree and 49 seconds for the benign chorea pedigree. This experiment was repeated 100 times for each pedigree.

Random downhill search performed considerably better than the greedy algorithm on both the EAST syndrome and benign chorea pedigrees. All runs found the optimal peeling sequences of 625 and 2145, respectively. In the worst case, the optimal sequence was found at iteration 3,096 for the EAST syndrome pedigree and at iteration 299,425 for the benign chorea pedigree.

Summary

For the pedigrees we have tested, random downhill search appears to be an appropriate method for generating a good peeling sequence. It is possible that for larger, more complex pedigrees, it will not perform as well. In this situation, a more advanced method like the simulated annealing algorithm detailed in Thomas' review [119] would be necessary. As the peeling sequence needs only be calculated once and read from a

file for the actual simulation runs, we did not investigate parallelism options. However, for larger pedigrees this might be warranted.

6.5 LOD scores

For LOD scores, we are concerned with measuring several quantities; these will be of use with the locus sampler as well. The first is to identify the granularity of parallelism possible on the CPU; we would prefer to parallelise the likelihood calculations themselves in a data parallel fashion, but would require this to be comparable in speed to the more task parallel workload of assigning one locus per thread. Secondly, we must answer the question posed in section 5.4.3 as to how much work should be done by each CUDA kernel.

6.5.1 CPU Thread Granularity

Questions

To make LOD score calculations multithreaded, there are two possibilities open to us: each thread could compute a complete LOD score (task parallelism) or each thread could compute a single genotype likelihood in each peel operation (data parallelism). If we could parallelise each individual peel operation, then this would be of enormous benefit to the locus sampler. The single-threaded locus sampler performs a scan of all loci in a pseudo-random order. If we need to run the locus sampler at different loci concurrently, then strictly retaining this order will be difficult.

We used the data from chromosome 1 of the EAST syndrome pedigree for this experiment. We calculated a single LOD score located halfway between each pair of markers. As chromosome 1 was typed with 780 SNPs, each measurement involved the calculation of 779 LOD scores. Measurements were taken as described in section 6.3.1. Experiments were performed using 1–4 threads as that was the number of CPU cores available on the test machine. Each data point is the average over 1,000 trials.

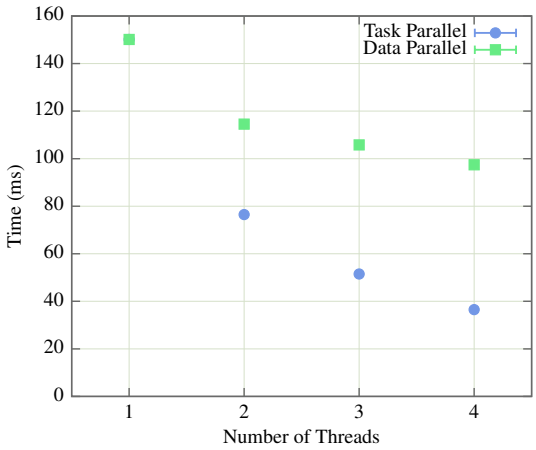


Figure 6.5: Timing measurements from calculating LOD scores for the EAST syndrome pedigree with varying numbers of threads. As the number of threads increase, finer granularity tasks (data parallelism) suffer slowdown from the overhead involved in keeping the threads busy.

Results

A graph of the results is shown in Figure 6.5. A single thread running either task or data parallel method are equivalent. From 2–4 threads, the overhead of keeping just two threads busy is apparent in the green data parallel experiments, with each additional thread providing less and less improvement.

The task parallel threads scale almost linearly. The work to do is coarse-grained enough that the overhead of distributing work to threads is negligible. There will be a limit to its ability to scale in this manner, but we did not have access to a processor with more than four cores to discover the asymptote in the graph.

Conclusions

For the CPU, tasks must be kept as coarse-grain as possible. If we do this, then compute-intensive tasks can achieve near linear scaling, at least across the narrow range of threads investigated. It might be possible to improve these results with the addition of SIMD instructions, providing practical fine-grain instruction-level data parallelism within threads. However, this would require an unknown level of code restructuring and we consider it outside of the scope of this thesis.

6.5.2 GPU Block Dimensions

Despite the results for CPU parallelism, the situation will be very different for the GPU, where we have custom hardware developed for the purpose of data parallel processing.

Questions

The main question we have to answer is: what is the granularity of tasks to give to each CUDA kernel? We could write a separate kernel per peel operation and ensure we use the same number of threads as there is work to do. This sounds like it would be efficient as we would never have any threads sitting idle. But this would require many round-trips to the CPU to call the next kernel and could create a non-negligible amount of overhead.

There is another reason this might not be the most efficient method and it is related to a concept we have not covered, *occupancy*. A graphics card is an embedded system with a finite amount of RAM. Whilst regular PCs only have a finite amount of RAM, this is abstracted away from the programmer with *virtual memory*. Virtual memory allows us to pretend we have the maximum possible amount of RAM, i.e. the complete address space. If we use more memory than the amount of physical RAM, it is swapped to the hard disk (albeit at a massive performance penalty). A GPU does not have the luxury of swap space, but even if it did we would not use it because we are focused on performance. Instead, the exact number of blocks and threads that can be supported by the total number of registers and shared memory are run at once and will cause an error if the block/thread dimensions exceed these constraints. Therefore, we may need 512 threads for a peel operation, but if the maximum number of threads that can be run is 768, then we will not be permitted to use the remaining 256. It would be more efficient to run three blocks of 256 threads or some other factor of 768.

For the above reasons, it may be more efficient to use a constant number of threads per block, but as each peel operation requires a different number, we need to ascertain what the correct number to use is without hurting efficiency.

To experiment with these ideas, we used the EAST syndrome pedigree to measure

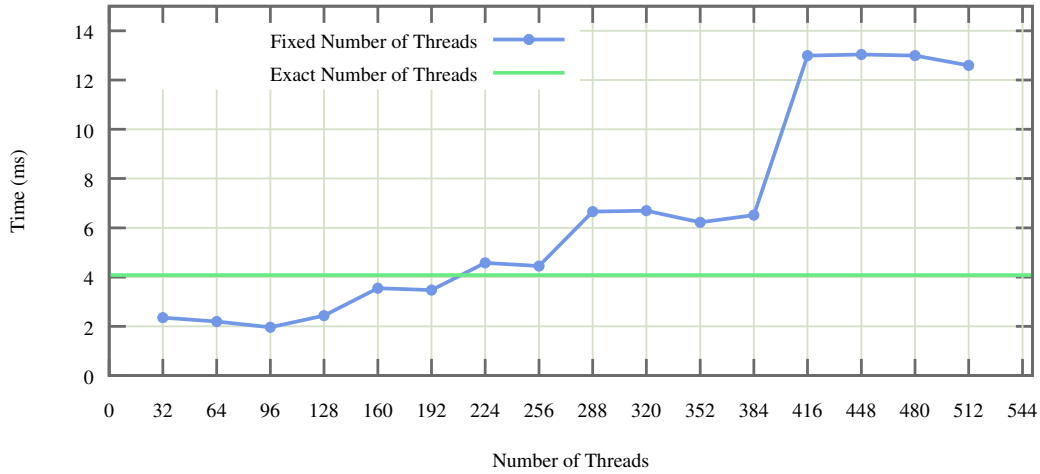


Figure 6.6: Timing measurements from calculating LOD scores for the EAST syndrome pedigree with varying numbers of GPU threads per block. As the number of threads increase, the overhead of threads idling causes slow down.

the time it took to compute LOD scores for the whole of chromosome 1. We calculated a single LOD score located halfway between each pair of markers. As chromosome 1 was typed with 780 SNPs, each measurement involved the calculation of 779 LOD scores.

Results

The graph in Figure 6.6 shows the results. All timing was performed using CUDA events and each data point is an average over 10,000 measurements. Standard deviation bars, omitted from the graph, were in the range of 0.02–0.08 ms. The green line indicates the time it took to run an independent kernel per peel operation, where each kernel was given the precise number of threads as there was work to do. As the EAST syndrome pedigree is composed of 27 individuals, this involved 27 separate kernel invocations for each set of 779 LOD score calculations. The blue line shows the timing measurements from using a fixed number of threads per kernel, where each kernel calculates a single LOD score which involves peeling the complete pedigree at a single location. Experiments were performed at increments of 32 threads, ranging from 32–512. The increment of 32 is necessary because, as we stated in Section 5.3.2 threads are run in quanta, called warps, of 32 threads. From the graph it is apparent that it is more efficient to perform the entire pedigree peel in a single CUDA kernel. The most

efficient number of threads to use for peeling the EAST syndrome pedigree is 96.

Conclusions

The optimal peeling sequence for the EAST syndrome pedigree has a maximum cutset size of 3, meaning there will be a maximum of 64 operations to be performed per peel operation. Selecting the precise number of threads as there is work to do performs comparatively poorly, due to it requesting 4, 16 or 64 threads per operation, which is always rounded up by the hardware to a multiple of 32, so what we see is mostly overhead.

Setting a fixed number of threads is more nuanced. High numbers of threads are slow because most of the threads are sitting idle. For middling numbers of threads, the graph has plateaus, e.g. 224–256 threads, which is due to the occupancy. Whilst there are more threads, the number of blocks resident in a multiprocessor is the same. The optimal number of threads is 96, but if the largest peel operation requires 64 calculations, then a minimum of 32 threads are sitting idle. What we believe is happening here is a side-effect: by forcing the scheduler to juggle more warps than necessary, it may reduce contention on the bus, increasing total throughput. The implication being, that even if we calculate the occupancy we expect from a given kernel, this is only a guide for its performance. The difference may be small, but it is in fact 10% faster to use 96 threads, rather than 64. The optimum number of threads will be highly data and hardware dependent. Therefore, pre-simulation, the application should always perform these tests, which are cheap in terms of time, in order to dynamically optimise itself for the rest of the simulation.

6.6 Locus Sampler

We have seen from the results in the previous section that the CPU code must be capable of running multiple loci concurrently to scale and the GPU code must do the same. For LOD scores, the calculation of each score was independent as it was only dependent on

```
begin
    window_length ← N
    locusOrdering ← {}
    descentGraph ← current_graph

    for i ← 1 to window_length do
        locusOrdering = locusOrdering ∪ {i}
    end

    random_shuffle(locusOrdering)

    for i ∈ locusOrdering do
        for j = i; j < number_markers; j+=window_length do
            run_locus_sampler(descentGraph, j)
        end
    end
end
```

Algorithm 3: Algorithm for the windowed locus sampler. If the window length is set to the number of markers then this degenerates down to a non-windowed (default) locus sampler. A parallel implementation of this sampler would involve parallelising the inner for loop, containing the call to run_locus_sampler.

the descent graph being scored. The locus sampler is creating new realisations of the descent graph based on flanking loci, so we need a way to run this in parallel without affecting the properties of the Markov chain.

6.6.1 Windowed Locus Sampler

Algorithm 3 shows the pseudo-code for the windowed locus sampler. A normal scan of the descent graph by the locus sampler involves all loci being sampled in a pseudo-random order. The windowed version proceeds in the same way, however, this time we only randomise the ordering of a single window and apply this to all windows in parallel. The windowed version of the locus sampler with a single window degenerates to the non-windowed original locus sampler.

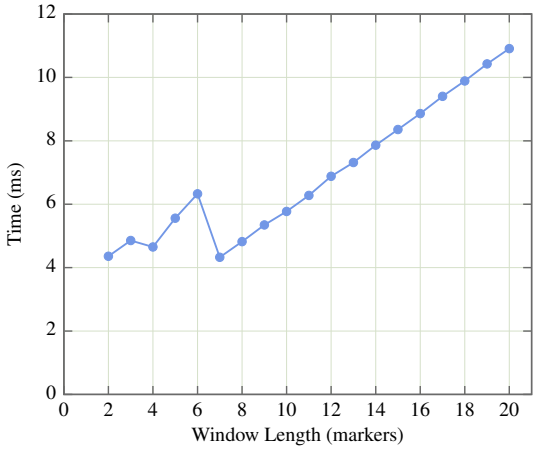


Figure 6.7: Variable numbers of markers per window affect the run-time performance of running a complete scan of the GPU version of the locus sampler.

6.6.2 Effect of Window Size on GPU Performance

Questions

Before we look at how the window size affects the performance of our sampling, we will look at how having a window will impact the GPU run-time performance of the locus sampler. A CPU parallel version should run with the number of windows set to the number of processor cores, but with the GPU we cannot be so coarse-grained and maintain high performance. This section investigates how small the window of markers need to be to achieve maximum performance.

Results

Figure 6.7 is a graph of the time required to perform a single scan of chromosome 1 of the EAST syndrome pedigree using the GPU version of the locus sampler. Each data point is the result of 10,000 repeated measurements. Standard deviations were omitted due to being uniformly small (largest being 0.2 ms).

From a window length of 7 markers the run-time rises linearly. Prior to 7 markers, these timing results are artefacts from the way CUDA schedules work to be done on our test graphics card. For example, for 780 markers, a window of 6 markers would incur 6 locus sampler kernel invocations of 130 markers, and a window of 7 markers would

incur 7 locus sampler kernel invocations of 112 markers. The GTX 580 graphics card that this test was run on has 16 multiprocessors and the locus sampler runs 8 concurrent blocks on each multiprocessor, i.e. 128 in total. Therefore, running 130 markers per invocation is wasteful because we run the first 128 concurrently and then need to wait for the last two before returning, resulting in this bump in the graph.

Conclusions

Even the current generation of GPU would require us to run exceptionally small windows of markers to be able to scale, between 2 and 10 markers. We have to investigate the effect this has on the mixing and convergence properties of the Markov chain. Longer runs of the chain might still permit convergence, but this would force us to make a parallelism / chain length compromise beyond which we could only get a speed-up with faster hardware. Despite potentially impacting the quality of the simulation, there is hope that the meiosis sampler will still be able to compensate.

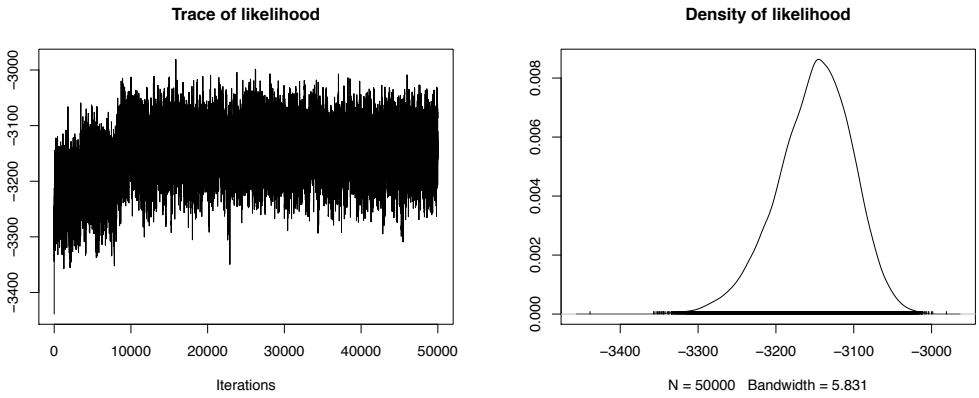
6.6.3 Effect of Window Size on MCMC Performance

Questions

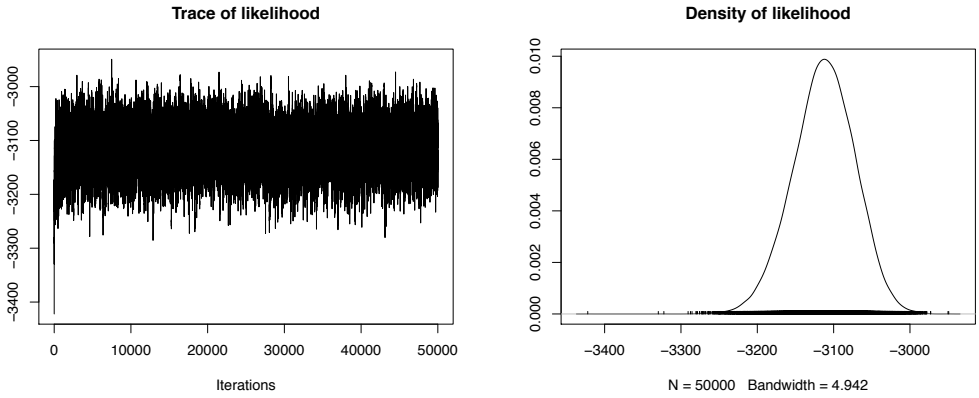
We know that to fully utilise even current generation GPUs, we need the window size to be small. In this section we have to investigate the effect this will have on the mixing and convergence properties of the Markov chain.

Diagnostics

Taking our running example of chromosome 1 of the EAST syndrome pedigree, we looked at three different windowing scenarios to assess their impact on the Markov chain. A single window of 780 markers, which is identical to the default locus sampler, eight windows of 98 markers, which is how a modern multicore CPU with eight cores would run the simulation, and, finally, 390 windows of 2 markers, which is how a GPU would be forced to run to achieve high utilisation. Using 2 marker windows is the



(a) 390 Windows, 2 Markers



(b) 1 Window, 780 Markers

Figure 6.8: Trace and density plots of windowed locus sampler running on the EAST syndrome pedigree. Running the locus sampler with many short windows appears to affect mixing detrimentally.

most extreme level of windowing and would be the equivalent of running every odd numbered marker, followed by every even numbered marker.

We ran each experiment for 50,000 iterations. At each iteration, either the locus sampler or the meiosis sampler was selected at random with equal probability. At each iteration a complete scan was performed of either all loci or all meioses. Each experiment was repeated 4 times. Unless otherwise stated, the 8 window experiment produced results indiscernible from the single window. Figure 6.8a is a trace of a typical 390 window experiment. Mixing is slower than with a single window (Figure 6.8b), but eventually converges to the target distribution.

The Gelman–Rubin diagnostic across the four repeated chains showed median val-

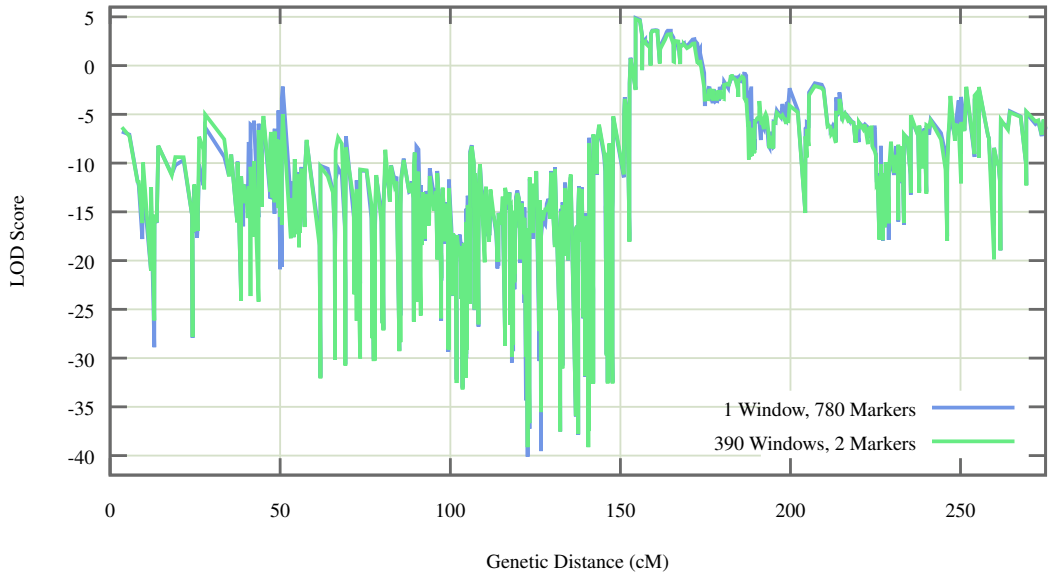


Figure 6.9: Linkage analysis of chromosome 1 of the EAST syndrome pedigree. Despite being shown to affect mixing, running 390 windows of two markers does not appear to affect the LOD curve versus the default locus sampler.

ues of 1.04, 1.13, 1.20 for the single window, 8 window and 390 window experiments respectively. Whilst the single window experiment appears to have converged, as the number of windows increase in number, this degrades.

Results

Despite some of the poor results from the diagnostics, what we are really interested in is the impact of multiple windows on the LOD curve. Figure 6.9 shows a graph of two anecdotal linkage analyses from the previous diagnostic runs. To generate these scores, the first 5,000 iterations were discarded as burn-in and the chain was thinned by scoring every 10^{th} iteration. All runs of the chain appeared similar to the one shown. The blue curve was run without any windowing (the standard locus sampler), whereas the green curve was run with 390 windows of 2 markers. The differences between the two runs are minor. Despite the problems suggested by the MCMC diagnostics, this did not translate into any real differences in the final result.

Conclusions

The MCMC diagnostics show the windowed locus sampler is less likely to converge than the original locus sampler. However, the results of the linkage analysis remained largely unaffected. Whilst we can see the point in maximising the size of the window, the smallest windows possible ensure a higher level of hardware utilisation and, at least from the perspective of the LOD scores, appears not to be detrimental.

6.7 Meiosis Sampler

Unlike the locus sampler, we cannot run multiple instances of the meiosis sampler simultaneously. Instead, we parallelise each meiosis sampler by exploiting the independence of $P(Y \mid S)$ at different markers, then perform the sampling in a single thread. This works well on the CPU, where each processor core is fast, but is too slow on the GPU.

6.7.1 Poor GPU Performance

In 10,000 measurements of the time taken by the GPU to perform a complete scan of all meioses on the EAST syndrome pedigree using the meiosis sampler, the mean time was 69.9 ms (standard deviation $\pm$ 0.83 ms). This makes the time taken to scan 36 meioses over 16 times greater than performing 780 complete pedigree peels. The relative ease of converting the locus sampler to the GPU architecture has now changed which sampler is the most time consuming on a large inbred pedigree. We have two options: split responsibilities for the different samplers between the GPU, for the locus sampler, and the CPU, for the meiosis sampler; or else use an approximation of the meiosis sampler to better fit the GPU architecture.

6.7.2 Window-based Approximation

In order to achieve similar run-time performance to the GPU locus sampler, we use a simple window-based approximation. Instead of sampling across all markers in the

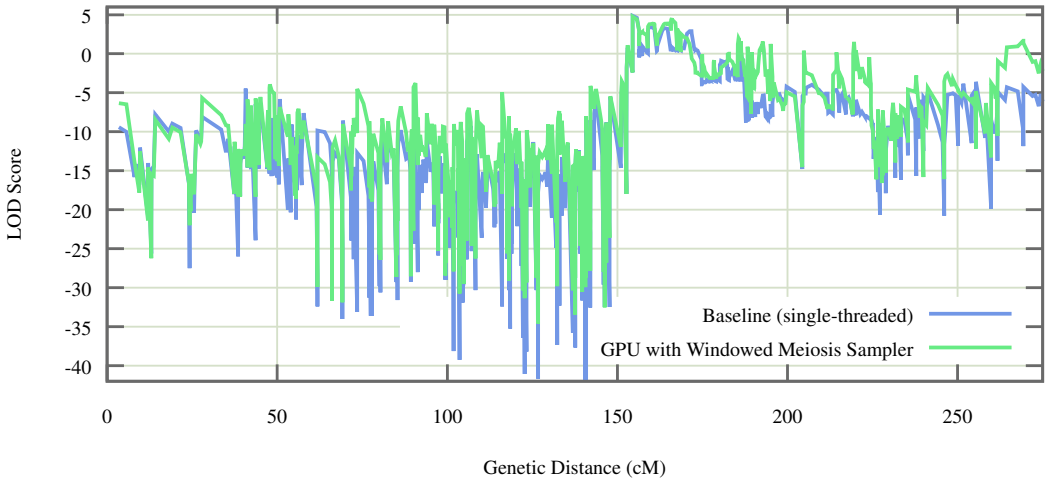


Figure 6.10: Linkage analysis of chromosome 1 of the EAST syndrome pedigree. The windowed meiosis sampler for the GPU appears to detrimentally inflate LOD scores compared to the single-threaded implementation.

chromosome for each meiosis, we sample across a shorter window of n markers. To try to mitigate the errors at the points where windows meet, we shift the window completely out of phase by $\frac{n}{2}$ markers every other meiosis being sampled. As during each scan, the meioses are sampled in a pseudo-random order, so which meioses are run in and out of phase is not fixed.

Setting $n = 32$, the meiosis sampler took 7.2 ms averaged over 10,000 scans (standard deviation $\pm$ 0.06 ms) per complete scan on chromosome 1 of the EAST syndrome pedigree. This puts the time taken for a complete scan of the meiosis sampler in the same range as the locus sampler on the GPU. As an example, we present a LOD curve in Figure 6.10 superimposed on the result from the single-threaded implementation. Additionally, we tried setting n to 64 and 128 markers. All these configurations produced similar results but with increasing run-times. Whilst the broad morphology of the LOD curve was maintained, the LOD scores are incorrect by several orders of magnitude throughout the chromosome. In the region of interest, however, where the LOD score is highest, the LOD curves are very similar. The impact this has on the results of several case studies is investigated in depth in the next chapter.

Several other windowing schemes were tried: with static windows of markers and partially overlapping windows, however, the results were inferior to the method de-

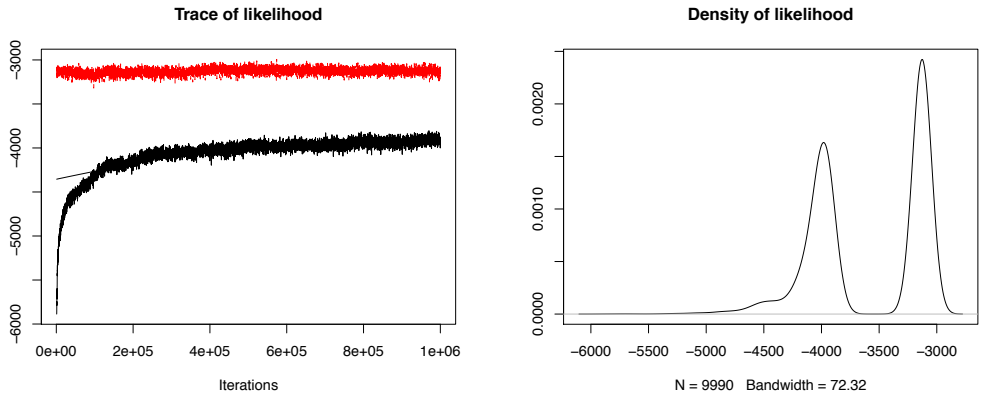


Figure 6.11: Trace and density plots of two long chain runs of 1,000,000 iterations The first 1,000 iterations were omitted for clarity and only every 100^{th} data point used. The black plot shows the single locus sampling method. The red plot shows the sequential imputation method.

scribed above.

6.8 Effect of Starting State

Questions

We have two main mechanisms to generate a legal starting state for the Markov chain: a single locus sampler and sequential imputation. Sequential imputation produces more likely starting states because it can take into account the probabilities of recombinations with flanking loci. More likely starting states, may help speed convergence, however, this might be counter-productive if we are unlucky enough to produce a state in a local minimum and the data is such that it always results in the chain getting stuck there. Single locus sampling produces highly dispersed points in the state space and could in principle avoid the potential problems of sequential imputation, but the cost of the additional convergence time may be prohibitive.

Results

Using chromosome 1 of the EAST syndrome pedigree, we ran two chains for 1,000,000 iterations, where each iteration is a complete scan of all loci or all meioses dependent on the sampler. At each iteration the probability that the locus or the meiosis sampler was

run is equal. Each chain was run in single-threaded mode on the CPU. We tested two methods for generating a legal starting state: the single locus sampler and sequential imputation. Each method is run 1,000 times with the state with the highest likelihood selected as the starting state.

The graph in Figure 6.11 shows both trace and density plots of the two runs; single locus sampling in black and sequential imputation in red. For the sake of clarity, the first 1,000 iterations were discarded as the likelihoods were so low that they made both graphs hard to read. Of the 999,000 remaining iterations, every 100^{th} likelihood was plotted.

It is clear from these results that even after a chain run of 1,000,000 iterations, the chain initialised with the single locus sampler had not converged. This plot gives us additional confidence in sequential imputation, as it clearly shows no additional modes were found in this example. The experiment was repeated several times, and these results were typical.

Conclusions

Past work has suggested that the single locus peeling approach has some merit where similar starting states are considered a problem, but for practical scenarios the additional convergence time required is so long we have not been able to successfully measure it. Subsequently, we use sequential imputation to generate a legal starting state for the Markov chain.

6.9 Scalability

We want to better understand the performance improvements parallelism achieves and how this scales with the number of processors we allocate to run a single analysis.

6.9.1 Amdahl's law

Amdahl's law helps us quantify how much we can improve the performance of a given system when only part of that system can be parallelised. Amdahl's law is stated as:

$$speedup = \frac{1}{(1-P) + \frac{P}{S}}$$

where P is the proportion of program run-time that can be parallelised and S is the speedup that can be achieved to the proportion P. An example of how this operates is that if a program takes 10 minutes to run, but 1 minute of that is sequential, then this is the limiting factor that dictates the best possible run-time (which would be 1 minute, assuming the work contained in the other 9 minutes is infinitely parallelisable).

6.9.2 Diminishing Returns

We used Amdahl's law to calculate what proportion of the programs run-time was reduced due to parallelism. Given that the speedup can also be calculated as

$$speedup = \frac{t_1}{t_2}$$

where t_1 is the time taken with 1 thread and t_2 is the time taken with 2 threads. We can calculate P with:

$$P = 2 \cdot \left(1 - \frac{t_2}{t_1}\right)$$

We calculated that the proportion of the locus sampler to be parallelised was 0.93 and the parallel portion of the meiosis sampler is 0.9 (these numbers were calculated for EAST syndrome chromosome 1 and will be a function of the complexity of the pedigree and the number of markers considered jointly). Assuming no additional overhead from distributing work to threads, we used Amdahl's law again to understand the diminishing returns of using more and more threads with different proportions of the two samplers (Figure 6.12).

Whilst there is some return all the way up to 128 processors, all the curves start to

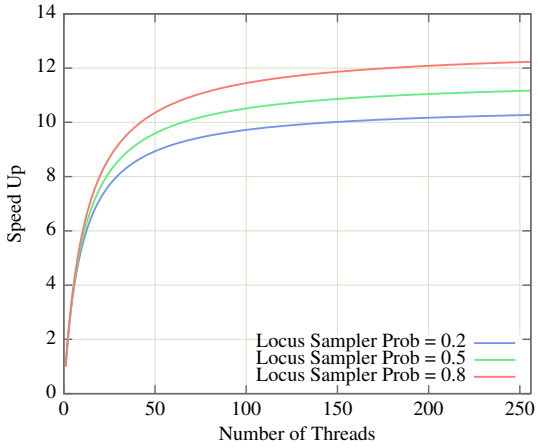


Figure 6.12: Theoretical speed up with different numbers of threads using Amdahl's law. The probability of using the locus sampler at each iteration affects the maximum speedup at the limit between 10x and 12x.

level off after that. We have shown 0.2, 0.5 and 0.8 probabilities of selecting the locus sampler at each iteration as these are the published extremes that will not affect the outcome of the results [35].

As linkage analysis is not trivially parallel, it is most efficient to use only a single thread. However, if we have an excess of processing power or really care about getting a particular result as quickly as possible, then we can use multithreading to achieve slightly greater than an order of magnitude speed improvement in the limit.

6.10 Summary

In this chapter, we looked at a series of benchmarks related to efficiently implementing Gibbs samplers for multipoint linkage analysis on parallel hardware. We will summarise them all below categorised by whether they are generic, related to multicore CPU or GPU implementations.

Generic

We showed that a random downhill search procedure greatly outperformed a simpler greedy algorithm, by always finding what seems to be the optimal peeling sequence. Generating a good peeling sequence can dramatically reduce the total amount of work

to be done by the locus sampler and calculating LOD scores. Whilst random downhill search proved suitable for both the EAST syndrome and benign chorea pedigrees, larger and more complex examples may require something that can more adequately explore the state space, for example, simulated annealing or other stochastic optimisation method. However, as the benign chorea pedigree is at the upper bound of the pedigree size a majority of practitioners would ever see in their career, we did not explore this further.

We investigated using different methods to generate the initial state for the Markov chain and found the single locus sampler unsuitable for the task at hand, i.e. consanguineous pedigrees with many markers. For this size of problem we can only recommend sequential imputation, which, for a small increase in complexity, permits far shorter simulation runs.

Multicore CPU

We showed that the multicore CPU versions of both the LOD scoring code and the locus sampler need to be run at concurrent loci as the overhead of parallelising the matrix computations did not scale even with the small number of threads used. We detailed a modified locus sampler called the windowed locus sampler and, based on multiple MCMC diagnostics, performed similarly to the default, non-windowed locus sampler.

Whilst we tested the smallest windows possible (2 markers) for the GPU, this also has implications for future multicore CPUs. Namely, the ability for the windowed locus sampler to scale to hundreds of processor cores will be limited by the asymptotic returns of parallelism (related to Amdahl's law) and thread-related overheads, not the number of loci in a window.

GPU

The parallel Gibbs samplers for the GPU were much harder to design. It is a very different architecture compared to a multicore CPU, intended for massively data parallel

applications. Therefore, we sought to understand the impact different design decisions had on the level of hardware utilisation and its interplay with the actual execution time.

We identified, counter-intuitively, that CUDA kernels for pedigree peeling should use a fixed number of threads per block, and that the optimum number of threads may be *greater* than the number of calculations to be performed. To facilitate the highest throughput, these parameters should be discovered empirically before the actual simulation is run.

Quite severe changes needed to be made to both locus and meiosis samplers, involving running different windows of markers, to get the most performance out of the GPU. For the locus sampler, this slowed the convergence of the Markov chain, but it continued to produce almost identical LOD score results. The changes needed for the meiosis sampler adversely affected the correctness of the LOD scores. Whilst we present results from the GPU implementation in the next chapter, we additionally present a hybrid application that utilises the GPU only for the locus sampler and calculation of LOD scores, which is not only competitive in terms of speed, but provides a level of accuracy that the pure GPU application is incapable of at the present time.

Conclusion

We now have an application for performing linkage analysis on large, consanguineous pedigrees with many markers that can use multiple threads on the CPU and/or a separate GPU. The multicore CPU code runs a user-specified number of threads and produces the same results as a single-threaded implementation. The manycore GPU code dynamically works out suitable parameters to best utilise the hardware at its disposal and gives more approximate results. We can also utilise both multicore CPU and manycore GPU concurrently to gain the advantages of both platforms running different aspects of the simulation.

These benchmarks have informed the design of how different modules function on different hardware platforms, but we do not have a feel for how this will translate to the complete analysis time. In the next chapter, we will take several complete linkage

projects and present not just analysis results, but end-to-end performance measurements of the total run-time.

7 Case Studies

I'd lay down my life for two brothers or eight cousins.

J.B.S. Haldane

The previous chapter was concerned with benchmarks related to the level of hardware utilisation different modules of SwiftLink achieves. In this chapter, we want to explore the impact our parallel Gibbs samplers have on both the run-time and the accuracy of linkage studies performed on real-world datasets.

We will look at three case studies: sensorineural deafness, EAST syndrome and benign chorea. Sensorineural deafness is an autosomal recessive phenotype, in this case, found in a small pedigree typed with many markers. Smaller pedigrees would normally be analysed with an implementation of the Lander-Green algorithm, i.e. Genehunter, Merlin or Allegro, and not an MCMC-based program, however, doing so permits us to see whether the accuracy of a trivial analysis is affected by increasing levels of parallelism and allow us to compare our results to the exact answer. The families with EAST syndrome and benign chorea are two large consanguineous pedigrees. For EAST syndrome, the causative mutation has already been identified, whereas in benign chorea, no analysis has yet been performed. Both are autosomal recessive phenotypes. With these larger pedigrees, we will look at the run-time performance and the LOD scores generated by SwiftLink as compared to the existing MCMC-based programs, Morgan and Simwalk. We will test three versions of SwiftLink: a CPU version that is capable of scaling to an arbitrary number of threads (*CPU SwiftLink*), a pure GPU version that we already know produces less accurate results (*GPU SwiftLink*), and a hybrid GPU

/ CPU version (*hybrid SwiftLink*). Hybrid SwiftLink leverages both platforms, taking advantage of whichever platform is faster for each sampler, running the locus sampler and LOD scoring code on the GPU and the meiosis sampler on the CPU. The chapter will end with a brief note comparing 32-bit and 64-bit performance of SwiftLink before a final discussion about all the case studies.

7.1 MCMC Parameters

One of the major drawbacks of MCMC, from a usability perspective, is that there are several parameters that must be set by the user or else the program must use overly conservative default values. There tends to be very little guidance as to how to set these parameters because they are data dependent and, even if they are set correctly, we can still be unlucky with non-deterministic algorithms.

In the experiments reported in this chapter, Simwalk (version 2.91) is always run with default parameters. As Simwalk can be very slow for large numbers of markers, we instead ran each chromosome in multiple batches of 50 markers, the size of which was chosen based on our experiences with several past projects. Further details are given for each individual case study.

For Morgan (version 3.03) and SwiftLink, we ran each simulation for a total of 100,000 iterations as recommended [89], comprising 10,000 iterations of burn-in and 90,000 iterations of simulation. At each iteration, either the locus sampler or the meiosis sampler was selected at random with equal probability. A single iteration comprised of a complete scan of all markers, in the case of the locus sampler; or all meioses, in the case of the meiosis sampler. We scored every 10^{th} iteration of the Markov chain. LOD scores were calculated halfway between each consecutive pair of markers. For initialisation of the chain, 1,000 runs of sequential imputation was performed, with the highest likelihood result used as the starting state. All analyses were performed on our test machine described in Section 6.1 that was otherwise idle. For CPU SwiftLink, we will always state the number of threads that were used. Hybrid SwiftLink was always

run with 4 CPU threads in addition to the GPU.

7.2 Data Preparation

In each of the reported case studies, datasets were preprocessed and formatted using Alohomora [105] (version 0.3) and Mega2 [91] (version 4.0). Mendelian inconsistencies were checked with PedCheck [95] (version 1.1) and familial relationships were checked by visualising them with GRR [2] (latest version 08/2003). GRR plots the mean and standard deviation of the proportion of genotypes that are identical-by-state between individuals. Pairs of individuals of a given relatedness cluster together, providing a check for the correctness of the pedigree. The gender of typed individuals was checked by looking at the level of hemizygosity in the X chromosome.

Modern SNP chips have many more markers than linkage analysis is capable of analysing without breaking the assumption of linkage equilibrium. To trim down the marker map for analysis, we use a python script called *SNP-butcher*. SNP-butcher reads in the marker map, minor allele frequencies and genotype data and outputs a sparser map containing SNPs that are no less than a user-specified genetic distance apart. It ignores SNPs with a minor allele frequency of 0.0 (at the population level) and SNPs where all genotypes are uniformly homozygous for the same allele and, therefore, uninformative.

7.3 Sensorineural Deafness

Sensorineural deafness is a kind of hearing loss that is most often caused by abnormalities in the hair cells found in the inner ear. It can be caused by damage incurred throughout the lifetime of the patient or can be inherited. Different patients can experience varying levels of severity from mild to total deafness.

Figure 7.1 shows a pedigree which is a single nuclear family containing eight siblings: three boys and five girls. Four of the children were affected (individuals 32, 34, 139 and 142). As there is not any gender bias and neither parent is affected, it is

suggestive of an autosomal recessive phenotype.

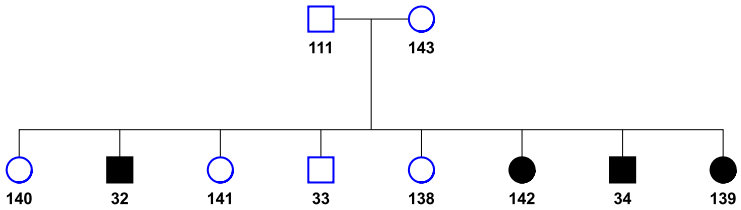


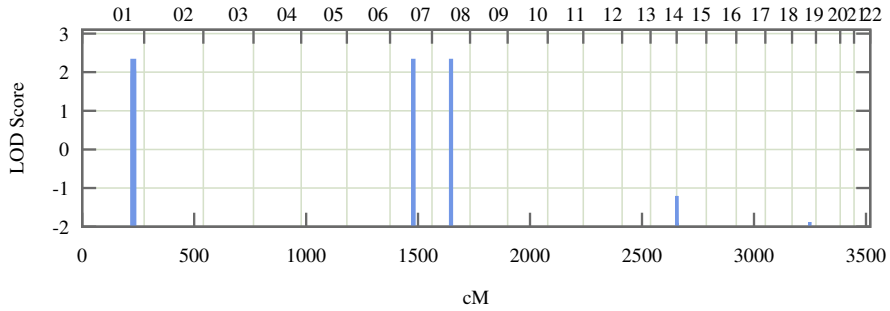
Figure 7.1: 14-bit pedigree with four affected children with sensorineural deafness. The segregation pattern of the affected individuals suggests this disease is an autosomal recessive phenotype.

For this case study, we are not concerned with any of the run-time characteristics of MCMC-based simulations. It is clear that where exact methods are applicable they should be employed. Not only are the results deterministic, but the analysis time will probably be shorter as well. Instead, we focus on the actual values of the LOD scores obtained and see how different approaches to parallelism affect accuracy.

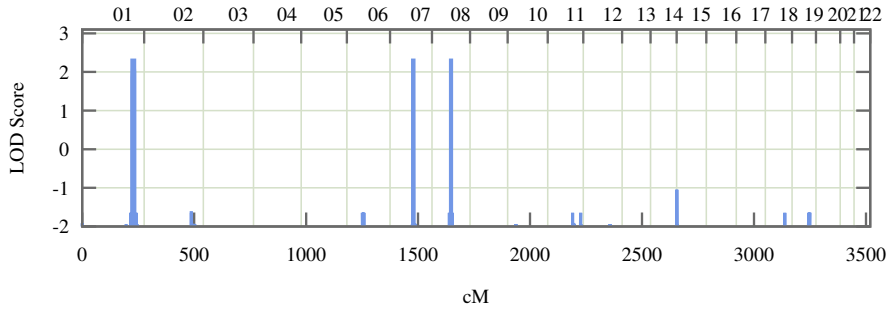
7.3.1 Methods

Genotyping was performed with Illumina human CytoSNP-12 300K SNP chips. All individuals, apart from individual 111, were typed. A genetic map for this SNP chip was unavailable, so inferred genetic distances from the HapMap project were used instead. Using SNP-butcher, 31,328 SNPs were selected that were no more than 0.05 cM apart and recombination fractions were sex averaged. The family was from Romania, so Caucasian allele frequencies were used. 14 SNPs were removed due to Mendelian errors detected by PedCheck. For parametric linkage analysis, a fully penetrant, autosomal recessive disease trait with an allele frequency of 0.001 was assumed.

For analysis we used the Lander-Green-Kruglyak-derived program Allegro (version 2.0f) [39], written by researchers at deCODE genetics. Allegro calculates exact LOD scores and scales better than Genehunter in terms of the complexity of the pedigree and the number of markers that the program can analyse jointly. We ran CPU SwiftLink with 1 and 4 threads, GPU SwiftLink and hybrid SwiftLink. All versions were run with the parameters described in Section 7.1.



(a) Allegro



(b) CPU SwiftLink (x1 thread)

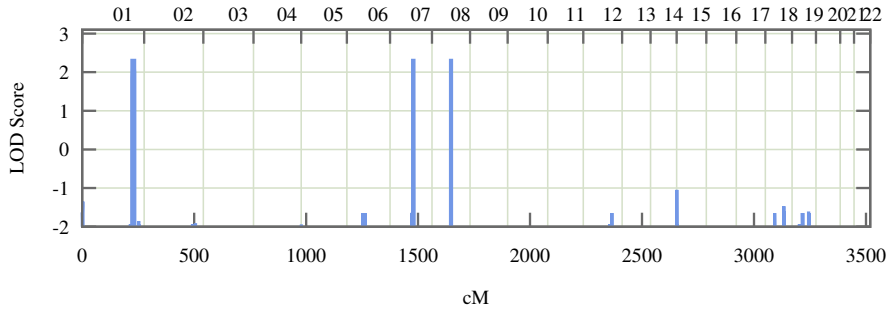
Figure 7.2: Genome-wide linkage scans for the sensorineural deafness pedigree performed by single-threaded applications.

7.3.2 Results

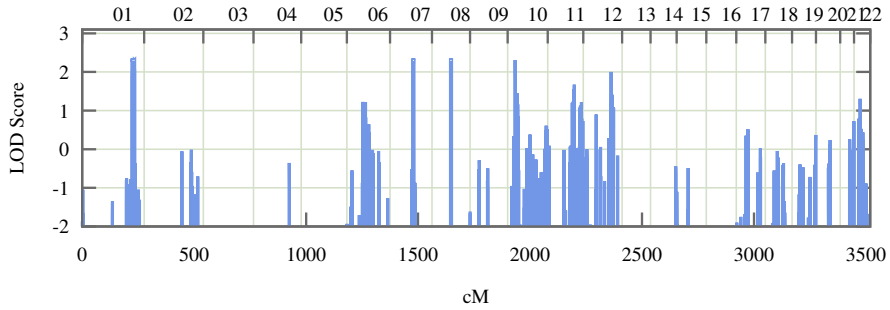
For this case study, the main aspects we wish to investigate centre around the accuracy of the MCMC-based simulations versus the exact LOD calculations calculated by Allegro. We first look at the LOD curves produced to compare the LOD scores and extent of the regions of interest identified by all programs before looking at the cumulative distribution of LOD scores.

LOD Scores

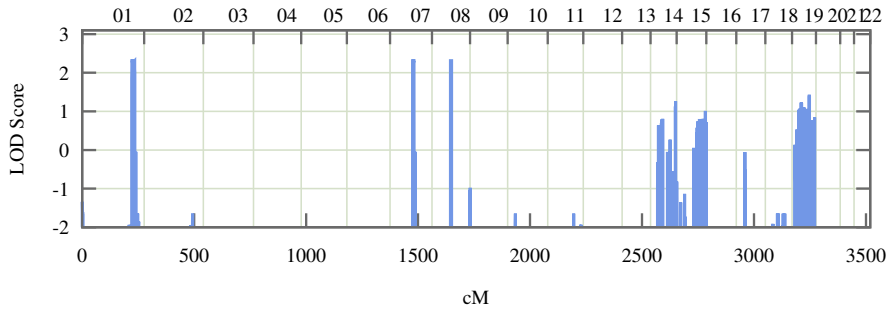
Allegro and single-threaded CPU SwiftLink produce much the same LOD curves (Figures 7.2a and 7.2b, respectively). The LOD scores of the most significant regions are the same (2.3059). Both programs found three identical loci of interest: chromosome 1 between rs7533652 and rs2769685, chromosome 7 between rs17302032 and rs2519637, and chromosome 8 between rs12541486 and rs7822888. The only differences between the two analyses are at LOD scores of -1 or lower, but, as LOD scores are on a log scale, these are relatively minor differences.



(a) CPU SwiftLink (x4 threads)



(b) GPU SwiftLink



(c) Hybrid SwiftLink (GPU / CPU x4 threads)

Figure 7.3: Genome-wide linkage scans for the sensorineural deafness pedigree performed by three parallel implementations of SwiftLink.

The three parallel implementations: CPU SwiftLink with 4 CPU threads, GPU SwiftLink and hybrid SwiftLink, are shown in Figures 7.3a, 7.3b and 7.3c, respectively. The multithreaded CPU SwiftLink performed almost identically to the single-threaded run. Both versions that utilised the GPU had additional false positives. For GPU SwiftLink, the number of false positives was extreme as it shows an additional peak on chromosome 10 with the same LOD score as the true regions of interest and a peak on chromosome 12 that is almost as high. Clearly, even for such a simple example, GPU SwiftLink can produce misleading results. Hybrid SwiftLink fared much better, but still produced many additional peaks on chromosomes 14, 15 and 19 with

LOD scores just greater than 1.

All three parallel implementations found what were considered the most significant regions by the single-threaded programs. The LOD scores of the most significant regions were either the same as Allegro (2.3059), in the case of CPU SwiftLink, or slightly different (2.3098), in the case of the two GPU applications. Minor differences in LOD score might be due to floating point representations being defined differently on the two architectures or even the order of operations not being quite the same.

Accuracy

So far we have only considered the regions of interest with the highest LOD scores and made a few statements broadly comparing the results. However, if we look at the cumulative distribution of LOD scores for each of the genome-wide scans (Figure 7.4), then we see some features that have been missed so far. The differences in the 90^{th} percentile are the most important from the perspective of the results shown in the LOD score graphs. All versions, including Allegro, are similar apart from GPU SwiftLink, which shows a large deviation from the others.

Excluding GPU SwiftLink, all other versions of SwiftLink are highly consistent throughout the cumulative distribution despite varying degrees of parallelism. This consistency is important because it tells us that, for the most part, parallelism had little impact on the results.

7.4 EAST Syndrome

EAST syndrome, described by Bockenhauer et al. [8], is a monogenic disorder related to improper renal tubular salt handling. Patients also have the following symptoms: infantile-onset seizures, ataxia and sensorineural deafness. Causative mutations were found in the gene KCNJ10 that encodes a potassium channel expressed in the brain, inner ear and kidney. The locus containing KCNJ10 was identified using linkage analysis and further narrowed down with haplotype reconstruction.

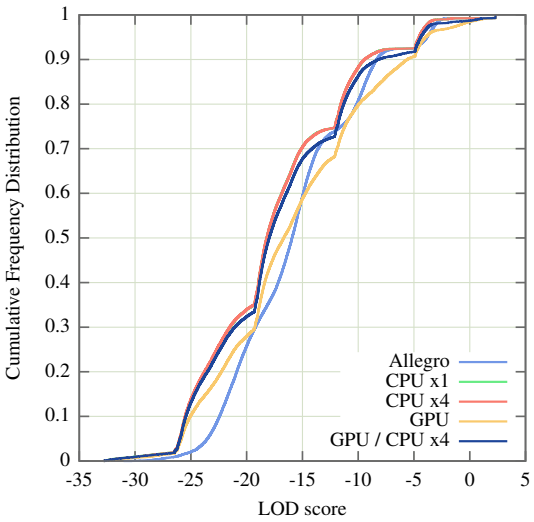


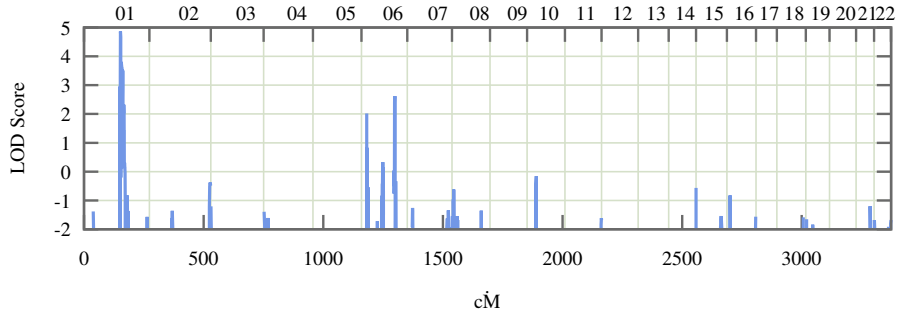
Figure 7.4: Cumulative distribution of LOD scores for the sensorineural deafness pedigree. Despite increasing levels of parallelism most of the versions of SwiftLink produce similar distributions.

The pedigree, previously shown in Figure 6.1a, contains 4 children all of whom presented with a similar phenotype. All other members of the pedigree were unaffected. All affected offspring are the result of first cousin marriages and share a common ancestor five generations earlier.

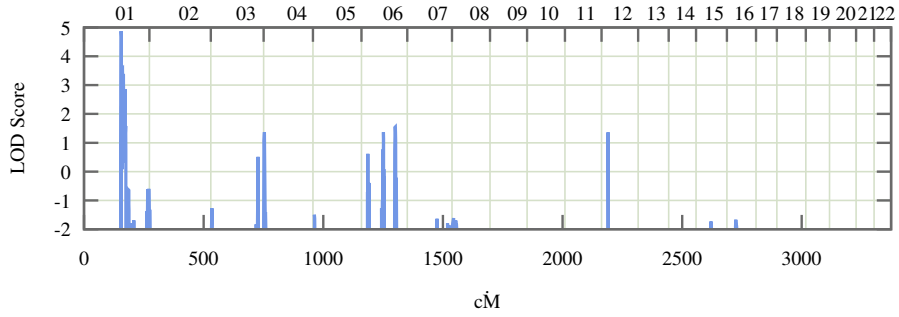
7.4.1 Methods

All 4 affected children and the parents of the 3 affected siblings were genotyped using GeneChip Human Mapping 10K SNP arrays from Affymetrix (individuals 821, 823, 824, 825, 1345 and 1347 in Figure 6.1a). Of the 10,204 autosomal SNPs typed, 14 were removed due to Mendelian errors detected by PedCheck. For parametric linkage analysis, a fully penetrant, autosomal recessive disease trait with an allele frequency of 0.001 was assumed. The family was of Pakistani origin, so the Asian allele frequencies from the deCODE 10K SNP map were used. Genetic distances were used from the deCODE map, the recombination fractions were sex averaged.

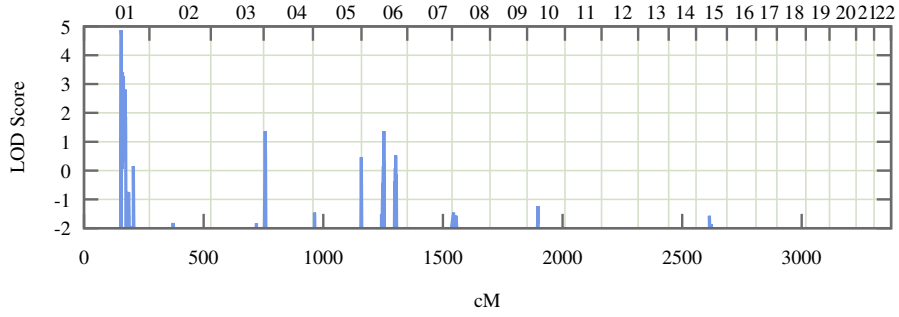
Whilst the original study utilised just Simwalk (version 2.91) to perform linkage analysis, here, we also ran Morgan (version 3.03) using the lm_linkage program (previously called lm_markers in older versions), CPU SwiftLink with 1, 2 and 4 threads, GPU SwiftLink and hybrid SwiftLink. Running Simwalk on the whole of chromosome



(a) Simwalk



(b) Morgan



(c) CPU SwiftLink (x1 thread)

Figure 7.5: Genome-wide linkage scans for the EAST syndrome pedigree performed by single threaded applications.

1 had a run-time of $\sim$42 days (60,935 minutes) on a 3.0 GHz Intel Xeon (results not shown). To make the run-time practical, we ran Simwalk in 212 independent fragments of 50 markers on UCL Legion, a computer cluster. Each fragment took $\sim$90 minutes, but total run-time was affected by the load on the scheduler and any cluster node failures. Therefore, the run-time of the Simwalk analysis is really a lower-bound. Morgan and SwiftLink were run with the parameters described in Section 7.1.

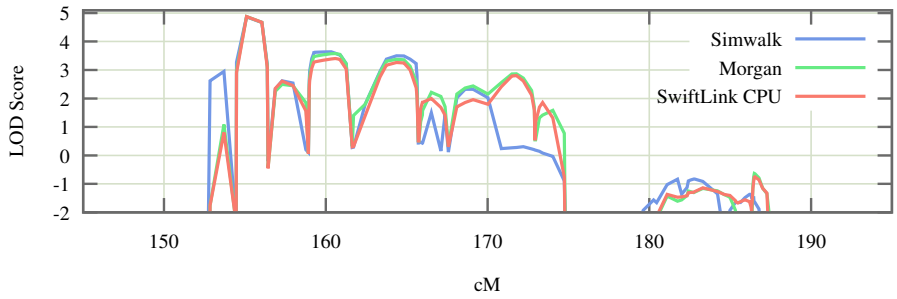


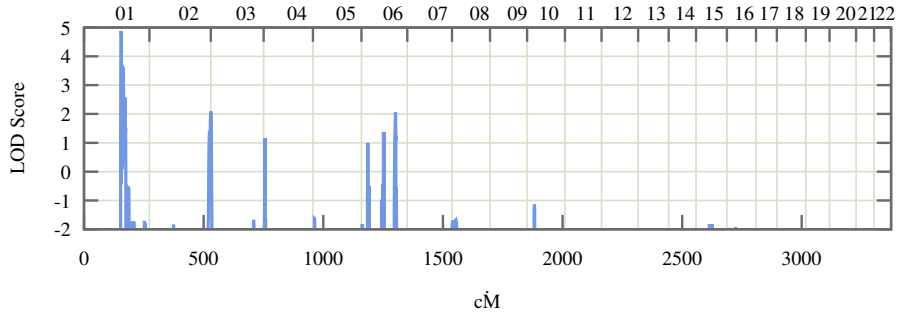
Figure 7.6: Region of interest from chromosome 1 of the EAST pedigree from all three single threaded applications.

7.4.2 Results

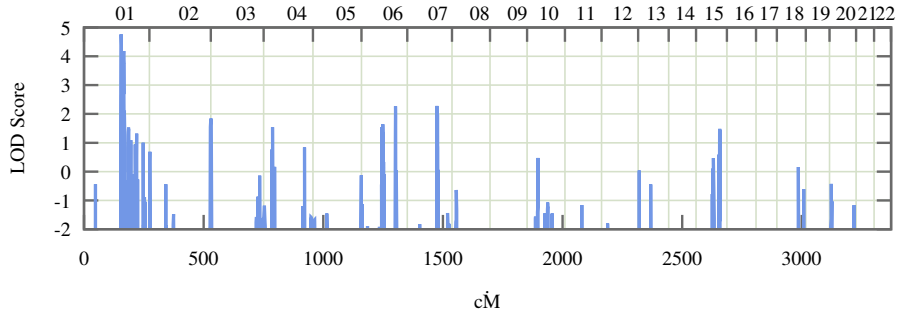
Comparing the output of different MCMC simulations is difficult because, whilst they should all converge to the same result in theory, there will be some variation between LOD scores even between multiple runs of the same program. Previous studies comparing MCMC linkage programs [126] are unsatisfying because they are compared only using examples that can be calculated exactly using either the Lander-Green or Elston-Stewart algorithm. What we want to know is whether our new parallel implementations give similar results to Simwalk and Morgan.

LOD Scores

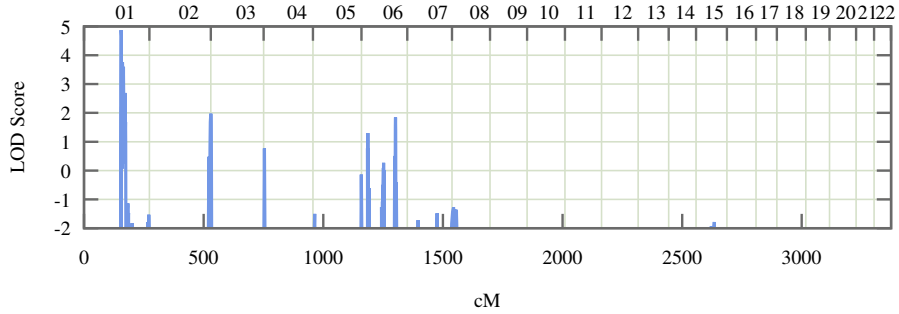
First, we will look at the single-threaded implementations. The genome-wide results for Simwalk (Figure 7.5a), Morgan (Figure 7.5b) and CPU SwiftLink (Figure 7.5c) have broadly the same morphology. Common across all three genome-wide scans, the most significant locus is on chromosome 1, with three less significant loci on chromosome 6. Each result has its own additional peaks, but these have at most a LOD score of $\sim$1 and do not affect what we would classify as a region of interest. For chromosome 1, the highest LOD score was 4.863, 4.874 and 4.875 for Simwalk, Morgan and SwiftLink, respectively. The region between 145 and 195 cM on chromosome 1 is shown in Figure 7.6. The most significant locus is at approximately 156 cM between flanking SNPs rs1891187 and rs1268524. The curves for each of the three programs are consistent with one another, with Simwalk showing some differences compared to Morgan and



(a) CPU SwiftLink (x4 threads)



(b) GPU SwiftLink



(c) Hybrid SwiftLink (GPU / CPU x4 threads)

Figure 7.7: Genome-wide linkage scans for the EAST pedigree performed by three parallel implementations of SwiftLink

SwiftLink, which are almost the same.

We present the results of three parallel versions of SwiftLink: CPU SwiftLink using 4 CPU threads (Figure 7.7a), GPU SwiftLink (Figure 7.7b) and hybrid SwiftLink (Figure 7.7c). The results of the multithreaded CPU SwiftLink are almost indistinguishable from the original single-threaded version, aside from an additional peak at the end of chromosome 2, however, artefacts such as these are routinely ignored as they can arise due to the lack of context at the ends of chromosomes. The similarity is further reflected in the finer details of the region of interest on chromosome 1 as well in Figure 7.8. GPU SwiftLink gives the messiest result, with the largest number of additional

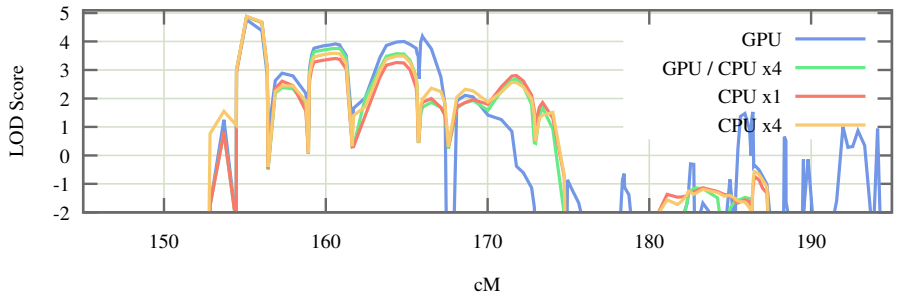


Figure 7.8: Region of interest from chromosome 1 of the EAST pedigree from all parallel applications.

peaks distributed all the way across the genome. The difference is even more obvious in Figure 7.8, where we just show the extended region of interest. The LOD scores frequently do not concur with the other programs. From $\sim$165 cM the LOD scores produced by different programs differed by more than an order of magnitude. This is due to the compromises that were necessary to improve the performance of the meiosis sampler. Hybrid SwiftLink appears to perform almost identically to single-threaded CPU SwiftLink, without the extraneous peaks genome-wide of GPU SwiftLink.

Irrespective of which application was chosen to run the EAST syndrome analysis, it would have resulted in the same outcome as all programs identified the correct locus containing the gene KCNJ10.

Performance

Table 7.1 summarises each complete analysis for all programs and Figure 7.9 shows the run-time for the analysis of each of the 22 chromosomes for all versions of SwiftLink. Simwalk and Morgan were omitted from Figure 7.9 as their run-times were so long as to obscure the differences between the different versions of SwiftLink.

Firstly, even the single-threaded version of SwiftLink ran 2.7x faster than Morgan for the complete analysis of all chromosomes and 20.8x faster than Simwalk, if we assume each fragment was run sequentially. For SwiftLink, the improvement from adding threads is sub-linear. Doubling the number of threads from 1 to 2 results in a 82% speedup, whereas 2 to 4 threads provides a 76% speedup. This is due to parts of the samplers that cannot be parallelised, which, for the CPU code, is mainly the sampling

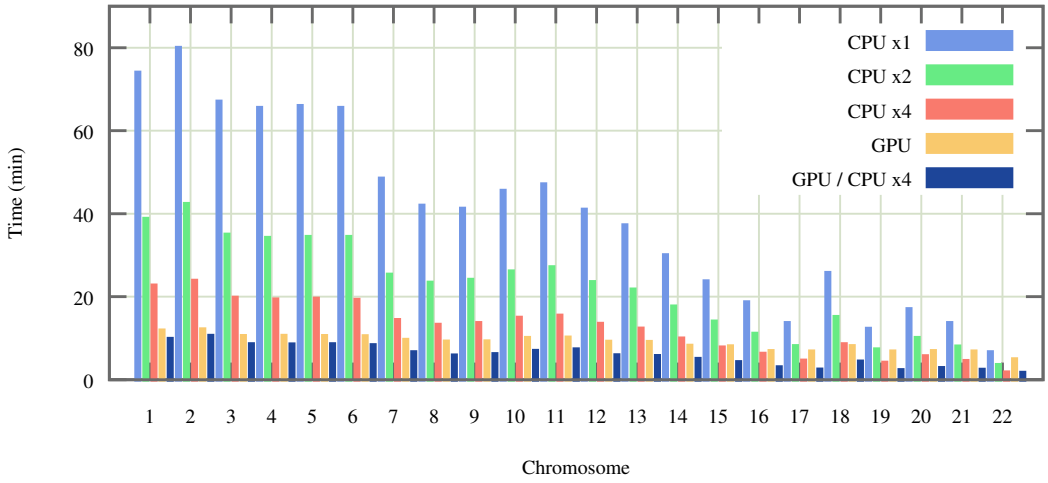


Figure 7.9: Run-time comparison of all versions of SwiftLink for each of the 22 chromosomes from the EAST syndrome pedigree.

phase of the meiosis sampler. These results concur with our theoretical analysis using Amdahl's law (Section 6.9).

GPU SwiftLink is, at best, 6.2x faster than the single-threaded CPU version. The level of improvement is proportional to the number of markers, the worst being chromosome 22 having the least markers, which is only 1.3x faster. The improvement from parallelism was almost uniform for the CPU code as the tasks involved were relatively coarse-grain, contrasted with the GPU that requires many markers to keep hundreds of CUDA cores occupied. However, taking the total run-time into account for all 22 chromosomes, the GPU was still 4.5x faster overall than the single-threaded CPU implementation.

The version with the best performance was hybrid SwiftLink. This is not surprising as we showed that the locus sampler was more amenable to being parallelised due to the large numbers of markers on the GPU, whereas the meiosis sampler functions best where there is still considerable individual processor speed. Hybrid SwiftLink takes advantage of both platforms to achieve high performance. Hybrid SwiftLink, similar to GPU SwiftLink, provides a greater speedup for longer chromosomes and is overall 7x faster than single-threaded CPU SwiftLink, the equivalent speedup of 16 CPU threads according to the theoretical analysis in Figure 6.12.

SwiftLink uses far more RAM than Morgan and Simwalk (see Table 7.1). SwiftLink's

Program	Max RAM Usage (MB)	Total Run-time (hours)
Simwalk	5.0	305.6
Morgan	10.0	39.9
CPU SwiftLink (x1 thread)	239.0	14.7
CPU SwiftLink (x2 threads)	239.0	8.1
CPU SwiftLink (x4 threads)	239.0	4.6
GPU SwiftLink	92.9 (+249.7*)	3.3
Hybrid SwiftLink	92.9 (+249.7*)	2.1

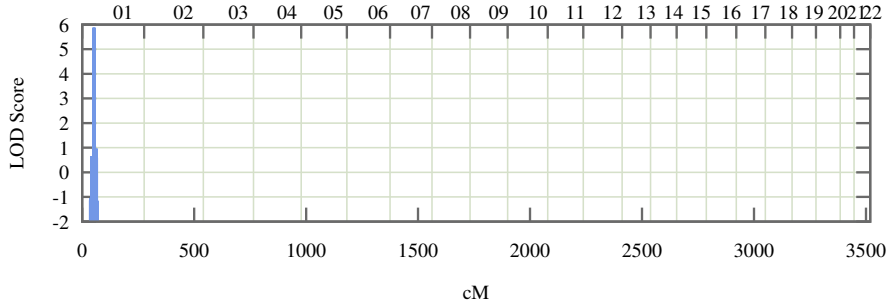
Table 7.1: Table of different program requirements for performing genome-wide linkage scans on the EAST pedigree. RAM measurements were made using the ps command.
(RAM required in device memory of GPU, in addition to host PC RAM requirements)*

high RAM requirements are due to it making extensive use of caching and requiring multiple copies of working data to avoid clashes from multiple threads. For example, the locus sampler maintains a copy of all of the intermediate peeling matrices for each and every marker, whereas Morgan will only have a single working copy. This is wasteful because we only ever need the number of working copies as there are active threads and, whilst not a major concern given the amount of RAM commonly found in a desktop PC, will be fixed in future versions.

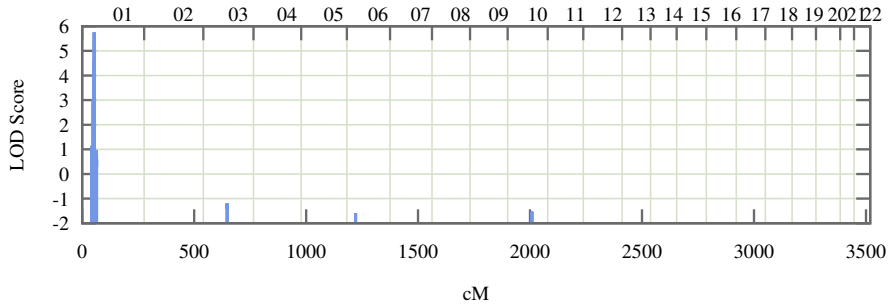
7.5 Benign Chorea

Chorea is a hyperkinetic movement disorder characterised by the presence of insuppressible involuntary movements. These spontaneous movements are predominantly distal (affecting head, face, tongue and distal extremities) and can range in severity from mild to disabling. Whereas Huntington's Chorea is a late onset neurodegenerative disorder marked by neurological decline and choreatic movements, so-called benign chorea is early onset and does not involve dementia. There are two known transmission patterns for benign chorea: autosomal dominant, with additional thyroid and lung problems; and autosomal recessive, without.

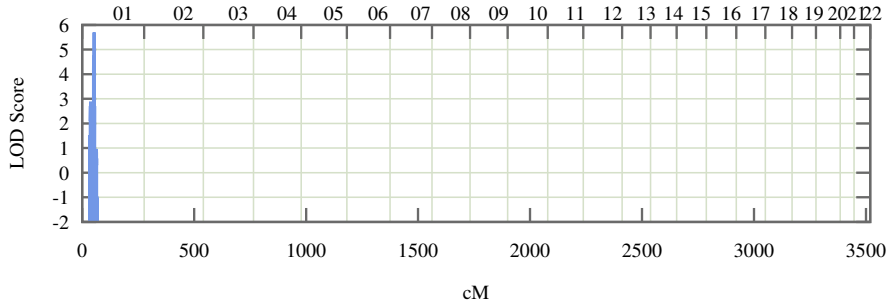
The pedigree, previously shown in Figure 6.1b, was first published by Poveda et al. [102]. There are 6 affected individuals in a 6 generation, 51 member, highly consanguineous pedigree originating from 3 founder couples. The mode of inheritance is



(a) Simwalk



(b) Morgan



(c) SwiftLink (CPU x1 thread)

Figure 7.10: Genome-wide linkage scans for the benign chorea pedigree performed by single threaded applications.

autosomal recessive.

7.5.1 Methods

Genotyping was performed with Illumina human CytoSNP-12 300K SNP chips on 19 individuals including all 6 affected patients. A genetic map for this SNP chip was unavailable, so inferred genetic distances from the HapMap project were used instead. Using SNP-butcher, 23,331 SNPs were selected that were no more than 0.05 cM apart and recombination fractions were sex averaged. Note that this is much lower than the number of SNPs used to analyse sensorineural deafness, despite using the same SNP

selection criteria. This is due to consanguineous pedigrees being more likely to contain uninformative SNPs. The family was from Columbia, so the Caucasian allele frequencies were used. 45 SNPs were removed due to Mendelian errors detected by PedCheck. For parametric linkage analysis, a fully penetrant, autosomal recessive disease trait with an allele frequency of 0.001 was assumed.

In the same way as we did for EAST syndrome, we ran genome-wide linkage scans using Simwalk, Morgan and SwiftLink. Simwalk was run in 487 independent fragments of 50 markers on UCL Legion. Each fragment took just over 4.5 hours to run once it had been scheduled. Morgan and SwiftLink were run with the parameters described in section 7.1. We ran CPU SwiftLink using 1, 2 and 4 CPU threads, in addition to GPU SwiftLink and hybrid SwiftLink.

7.5.2 Results

Similar to our investigation of EAST syndrome, we want to compare the results and performance of SwiftLink at increasing levels of parallelism with Morgan and Simwalk. Unlike previously, we do not have the ground truth, i.e. we do not know *a priori* what gene the causative mutation is located in.

LOD Scores

The three single-threaded genome-wide linkage scans in Figure 7.10 appear to be almost the same. There is significant linkage on chromosome 1 only, with maximum LOD scores of 5.820, 5.757 and 5.699 for Simwalk, Morgan and CPU SwiftLink, respectively. Looking at just the region of interest on chromosome 1 in Figure 7.11, some of the differences between the three programs become apparent. Firstly, all three programs have identified the 194 Kbase region at ~ 53 cM bounded by the markers rs2007451 and rs396648. SwiftLink and Morgan have identified a wider 691 Kbase region to the left of the leading peak with a maximum LOD score of 4.7 bounded by markers rs12759173 and rs448357. Simwalk did not find this region, this may have been due to the boundary between independent marker fragments being at ~ 52.5 cM.

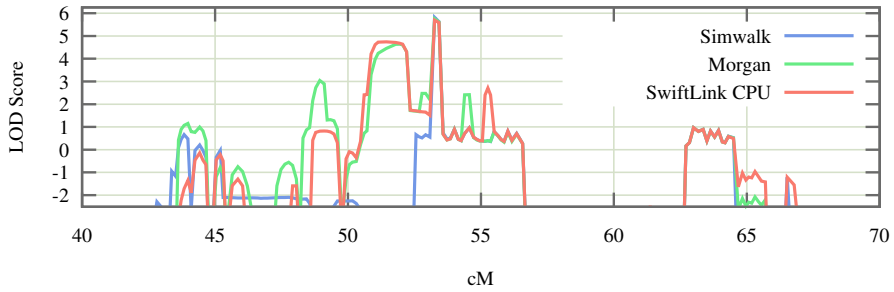


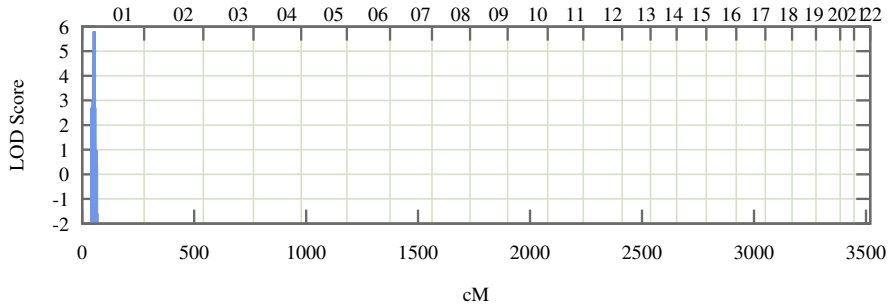
Figure 7.11: Region of interest from chromosome 1 of the benign chorea pedigree from all three single-threaded applications.

Neither of these regions contain any RefSeq genes, however, both contain spliced ESTs indicating the presence of, as yet, unnamed genes.

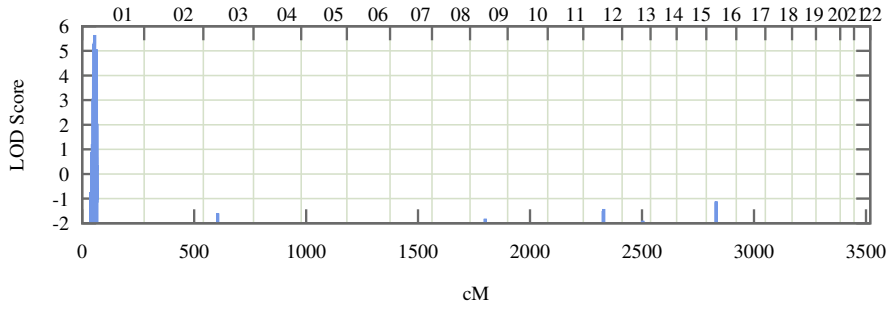
From the perspective of the genome-wide scans, all parallel implementations of SwiftLink produce similar results. Multithreaded CPU SwiftLink (Figure 7.12a), GPU SwiftLink (Figure 7.12b) and hybrid SwiftLink (Figure 7.12c) appear similar to the single-threaded analyses, with the most significant peak on chromosome 1. The maximum LOD scores were 5.806, 5.291 and 5.812 for CPU, GPU and hybrid SwiftLink, respectively. The region of interest is shown in Figure 7.13 comparing the LOD curves with the results from single-threaded CPU SwiftLink. Using multiple CPU threads produced much the same results as the single-threaded implementation. There are some differences upstream where the LOD score varies by two orders of magnitude, but these are limited and even at their peak are still three orders of magnitude lower than the maximum LOD score. This does not seem like an important difference because Morgan gave similar results (Figure 7.11). Hybrid SwiftLink showed several more extreme examples of this, such that the two highest peaks now have the same LOD score. GPU SwiftLink underestimates the LOD score for the leading peak and comes to the conclusion that the most likely location of the disease trait is elsewhere.

Performance

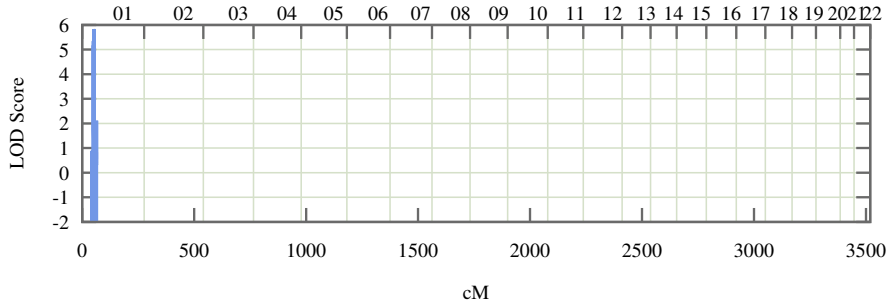
Figure 7.14 shows the run-time for the analysis of each of the 22 chromosomes and Table 7.2 summarises each complete analysis. The single-threaded version of CPU SwiftLink ran 2.2x faster than Morgan and 24.3x faster than Simwalk for the complete



(a) SwiftLink (CPU x4 threads)



(b) SwiftLink (GPU)



(c) SwiftLink (GPU / CPU x4 threads)

Figure 7.12: Genome-wide linkage scans for the benign chorea pedigree performed by three parallel implementations of SwiftLink

genome-wide analysis. Doubling the number of threads from 1 to 2 results in a 79% speedup, and from 2 to 4 threads, a 66% speedup. These results are not the same as the EAST syndrome analysis because the impact of the sequential sampling code is greater, due to us using more markers and the pedigree being larger. Despite these issues, hybrid SwiftLink is actually faster relative to the single-threaded version than it was analysing the EAST syndrome pedigree at 7.9x faster (compared to 7x faster for the EAST syndrome case study).

SwiftLink's memory requirements to analyse the benign chorea pedigree massively outstrip what was required in the EAST syndrome analysis (Table 7.2). Despite hav-

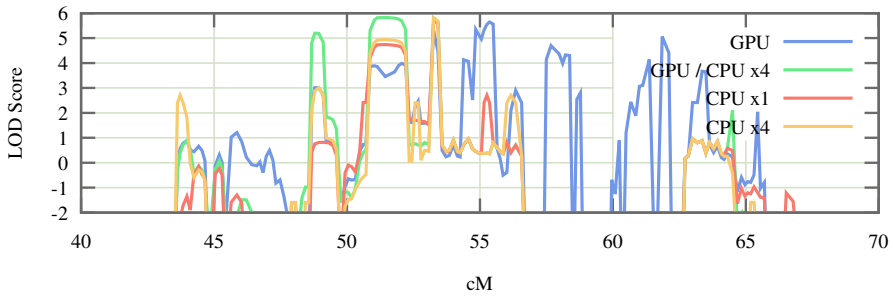


Figure 7.13: Region of interest from chromosome 1 of the benign chorea pedigree from all parallel implementations of SwiftLink.

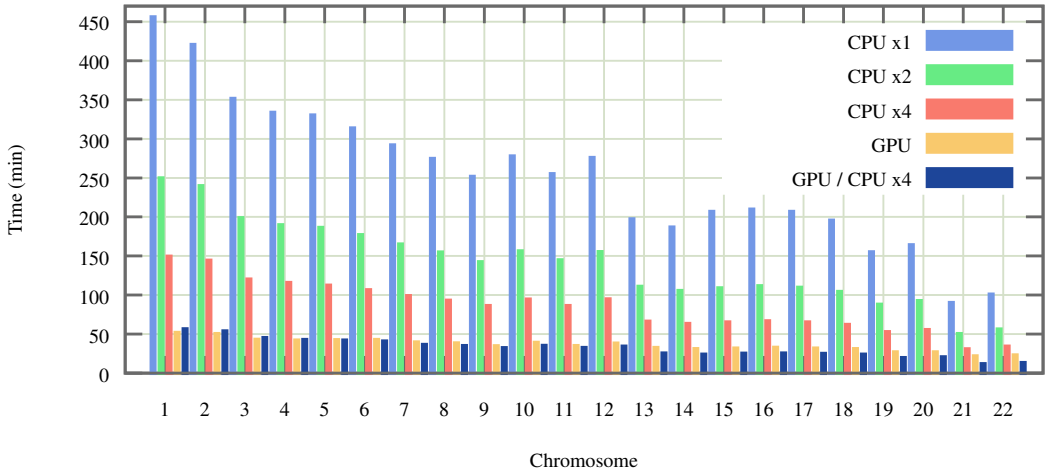


Figure 7.14: Run-time comparison of all versions of SwiftLink for each of the 22 chromosomes from the benign chorea pedigree.

ing the largest memory requirements previously, they did not seem unreasonable for a modern desktop PC, however it is now clear that it does not scale well. This will be addressed in future work. The GPU version should use as much memory as possible to keep the occupancy high and it is less of an issue because we assume the simulation is the only thing the GPU is running.

Caveat

Figure 7.15 shows a trace plot for CPU SwiftLink running single-threaded on the benign chorea pedigree. The simulation parameters were the same as in all previous experiments (see Section 7.1). Clearly, even the single-threaded analysis did not converge properly. This explains some of the differences in Figure 7.11 between Morgan and CPU SwiftLink. Whilst this does not excuse the worse performance of hybrid

Program	Max RAM Usage (MB)	Total Run-time (hours)
Simwalk	5.2	2249.8
Morgan	27.4	205.9
CPU SwiftLink (x1 thread)	2380.8	92.4
CPU SwiftLink (x2 threads)	2380.8	51.6
CPU SwiftLink (x4 threads)	2380.8	31.1
GPU SwiftLink	369.8 (+551.2*)	13.1
Hybrid SwiftLink	369.8 (+551.2*)	11.7

Table 7.2: Table of different program requirements for performing genome-wide linkage scans on the benign chorea pedigree. RAM measurements were made using the ps command. *(* RAM required in device memory of GPU, in addition to host PC RAM requirements)*

SwiftLink, it suggests there is more work to be done to improve the mixing of the Markov chain at least in this example. The benign chorea pedigree thwarts both samplers' abilities to mix in different ways. Recall, the locus sampler mixes poorly with tightly linked markers, which we certainly have in this situation, and the meiosis sampler has mixing problems when there are many untyped pedigree members. We cannot reduce the number of untyped members of the pedigree, but we can reduce the number of markers. Unfortunately, reducing the number of markers does not help as much as we had hoped, $\sim 10,000$ markers (asking SNP-butcher for a marker every 0.25 cM) suffers from the same problems and $\sim 6,000$ markers (every 0.5 cM) only appeared to converge half of the time.

Possible solutions to poor mixing include better samplers or multiple chain schemes like parallel tempering. Morgan implements the multiple meiosis sampler, but does not output the necessary information to diagnose non-convergence as SwiftLink does, so we did not test it. Parallel tempering may offer a suitable solution to improve the mixing of the existing samplers that we use and creates an alternative vector for employing parallel processing. A third idea, is that maybe the initial optimisation procedure needs to be improved. Previously, with the EAST pedigree, we saw that initialising the chain with single locus peeling produced a similar looking trace plot to Figure 7.15 (see Figure 6.11), but that initialising the chain using sequential imputation sped convergence. A better optimisation procedure for generating the initial state of the chain may be necessary to speed convergence in larger pedigrees such as the benign chorea pedigree.

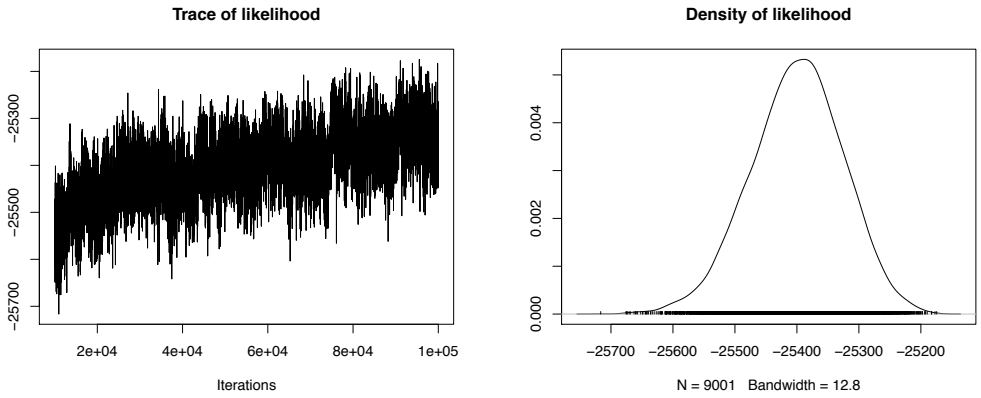


Figure 7.15: Trace and density plots for the benign chorea case study, with 10,000 iterations omitted as burn-in and thinned by only plotting every 10^{th} iteration. This is from the single-threaded implementation of SwiftLink, showing that even without parallelism being used the chain has not converged.

7.6 Performance of 32-bit and 64-bit Executables

An important detail that has not been mentioned is that in all the experiments that used just the CPU, SwiftLink was compiled into a 64-bit executable and in all experiments that used the GPU it was compiled into a 32-bit executable. SwiftLink can be compiled for either word length, but both had different performance characteristics. With the CPU, the 64-bit version is 15% faster than the 32-bit version because it permits the compiler to take advantage of longer instructions and additional registers. The GPU code, however, performs better in 32-bit mode because being able to use 32-bit pointers instead of 64-bit ones reduces register utilisation and increases occupancy. Additionally, there appears to be a bug in CUDA version 4.0 that makes large memory allocations far slower in 64-bit mode than 32-bit mode (at times dwarfing the run-time of the simulation). Unfortunately, the 64-bit CPU code cannot be linked with 32-bit CUDA code, therefore for hybrid SwiftLink, a 32-bit executable was used.

7.7 Discussion

From the case studies we have run, we have gained a feeling for how SwiftLink behaves in the real world, allowing us to better understand how parallel linkage analysis can

best be used. We can see from running the multithreaded version of SwiftLink, that parallelism has little effect on the accuracy of the LOD curves up to four CPU threads. However, it is clear from the hybrid GPU / CPU version that at the limit (i.e. many windows containing two markers each for the locus sampler) it starts to have an impact on accuracy in the form of inflated LOD scores in the EAST syndrome pedigree.

Even though we have demonstrated that SwiftLink scales across different numbers of CPU threads, it is clear that adding threads yields diminishing returns. This is not unexpected, as we know there are aspects of the simulation that could not be parallelised and Amdahl's law shows that this results in increasing underutilisation as we use greater numbers of resources. This means that we could analyse both EAST and benign chorea pedigrees quicker by running four instances of the single-threaded CPU SwiftLink on four different chromosomes. However, in the case that a single chromosome needs to be analysed or where we have a large excess of processors, multithreading is beneficial.

Whilst the results from the EAST pedigree (Figure 7.5) showed no benefit from running the entire chromosome in a single analysis compared to the windowed approach necessary with Simwalk, the benign chorea pedigree (Figure 7.10) exposed the worst-case scenario where a LOD score peak of 4.7 was missed because it was at the boundary between two windows. As a result, we do not recommend using Simwalk for any analyses with a large number of markers. Of course, Simwalk has one advantage over Morgan and SwiftLink, in that it is also capable of performing haplotype reconstruction. We recommend that linkage analysis be performed with SwiftLink and Simwalk used to narrow down a much smaller targeted region with haplotype reconstruction. Even though windows of markers makes Simwalk capable of utilising many processors with each window being independent, the time for each fragment ($\sim$1.5 hours in the case of EAST, $\sim$4.6 hours for benign chorea) means you would need a massive number of spare processors to achieve high performance.

We hoped that the windowed meiosis sampler on the GPU would have performed better than it did, but it produced poor results in all the cases studied. This did not matter too much, as the parallel CPU implementation of the meiosis sampler was faster

anyway, and hybrid SwiftLink, despite having problems with the complexity of the benign chorea pedigree, performed better in terms of accuracy and speed. At the start of this project, we felt that it was important to have as much code running on the graphics card as possible in order to achieve the highest performance. However, it is clear that, whilst certain algorithms can be successfully accelerated, others cannot. Contrast the meiosis sampler with the LOD scoring code that can exploit both coarse task parallelism and fine-grain data parallelism simultaneously. The lesson that can be learnt is that a combination of heterogeneous computer architectures can benefit run-time, trading off different characteristics in different situations.

From these experiments, the main conclusion we have come to is that, to ensure the accuracy of the analysis, the Markov chain should only be run on the CPU. Whilst multiple threads do not have an impact on the results up to four threads, we should investigate what the limit is with respect to the complexity of the pedigree and the number of markers used in an analysis. The GPU, however, should not be completely discounted as it performs exceptionally well calculating LOD scores. In our current application the Markov chain is thinned by only scoring every 10^{th} iteration and LOD scores are only calculated at a single point between each consecutive pair of markers. The next version of SwiftLink will be able to calculate LOD scores at an arbitrary number of points in between each marker. If this is performed at every iteration, then it is potentially many times more work than the actual simulation being run. SwiftLink will offload all LOD score calculations to the GPU which can then be handled at the same time as the CPU calculates the next state in the chain. We predict this will be even faster compared to a single-threaded implementation and improve the accuracy of results as more samples will be taken into consideration.

7.8 Summary

In this chapter, we took three complete linkage studies: sensorineural deafness, EAST syndrome and benign chorea. The sensorineural deafness case study was used to inves-

tigate the accuracy of our MCMC-based linkage analysis compared to the exact LOD scores calculated by the Lander-Green algorithm. We showed that all versions of our program, with the exception of GPU SwiftLink, produced accurate LOD scores and the same regions of interest. We saw that the parallel versions of our samplers either experienced little (hybrid SwiftLink) or no difference (multithreaded CPU SwiftLink) in the overall LOD curves generated at increasing levels of parallelism.

The two larger case studies: EAST syndrome and benign chorea, were run in Simwalk, Morgan and SwiftLink. For SwiftLink, we ran it with varying numbers of CPU threads, the GPU and both platforms combined. We showed that the multithreaded CPU version produced very similar results to the single-threaded programs and achieved a significant speedup, however, the sequential parts of the program made the gains of adding more threads sub-linear. In terms of performance, the greatest speedups were achieved by utilising both CPU and GPU thus leveraging the strengths of both platforms by running the samplers on whichever platform was best suited to the nature of the calculations required. Unfortunately, this is at the expense of accuracy in the regions of interest in a pedigree the size and complexity of the benign chorea example.

In the final chapter, we will conclude, making a critical appraisal of the thesis as a whole and outlining directions future research might take from here.

8 Conclusion

Good judgment comes from experience, and experience comes from bad judgment.

Frederick P. Brooks

In this final chapter, we will conclude by summarising the thesis, offer a critical evaluation of our work and outline the future research that can be carried out from here.

8.1 Summary of Thesis

The core motivation for this thesis was practical in nature, related to our involvement with a bioinformatics core facility providing expertise in genetic linkage analysis to institutes and clinicians throughout University College London. We noticed that, whilst a majority of projects could be performed by programs that used exact algorithms, e.g. Genehunter, Allegro and Merlin, the remainder either had to be abbreviated by removing siblings or inbreeding loops (occasionally with disastrous results) or else had to be run using MCMC-based programs like Simwalk or Morgan. Our experiences with Simwalk and Morgan were generally quite poor. Whenever Simwalk was used for a project, it would block all other work on the meagre resources we had locally. Morgan was always a pain to use as there was no readily available software to automatically create input files, which might be due to the specification of those input files changing with every release [1] and the number of markers it was capable of analysing jointly had previously been hard-coded to quite a low number. Our experiences with cluster

[1] Or at least it did during this thesis for versions 2.9, 3.0, 3.02 and 3.03!

computing had been mixed as well. Using a cluster requires an advanced knowledge of UNIX and running jobs can be frustrating. You do not get any results from jobs when the nodes they are running on fail and the scheduler exhibits bursty behaviour due to jobs that request a large number of resources. Our main frustration was that even if we invested in more hardware, we could not use more than 22 processors for a single genome-wide linkage analysis (this assumes we are not using Simwalk, which has to be run in many batches of markers to ensure the run-time is tractable, but then this suffers from the opposite problem in that we could never get enough processors all batches concurrently). What we wanted was the option to use an arbitrary number of processors for a single analysis. Namely, if we had analyses to perform, then no available computer processors should be sitting idle. As CPUs gain more processor cores in the future this problem will become more acute. To these ends, it was clear that we would need to fully understand the process by which Markov chain Monte Carlo was used to perform linkage analysis and to identify, within this process, independent calculations that could be performed in parallel.

In this thesis, we summarised the basic details of molecular genetics that someone with a background in computer science, for example, would find necessary in order to understand why we model inheritance the way we do. The hope was to promote an intuitive understanding of the samplers by detailing the biological processes involved. We stated the necessary likelihood calculations to assess whether two traits are in linkage and identified that naïve solutions would scale exponentially in terms of both the number of meiosis in the pedigree and the number of markers considered jointly. The main algorithms used to evaluate this likelihood and the basic scaling properties they exhibit were investigated. The Elston-Stewart algorithm scales exponentially with the number of markers, but linearly with the number of meioses in an outbred pedigree. Hidden Markov models in the form of the Lander-Green algorithm scale in the opposite manner, i.e. linearly with the number of markers, but exponentially with the number of meioses in a pedigree. Whilst the Lander-Green algorithm cannot analyse large families, the

amount of work that went into incrementally increasing the size of problem it could analyse speaks volumes for how important it is to gene mapping efforts. Outside of the historical context, the reasons for developing these different algorithms are not obvious, but when one considers that early studies would have only needed to assess linkage between two or three traits, the Elston-Stewart algorithm is clearly sufficient. Not until the seminal work of Botstein et al. on the potential of molecular polymorphisms for mapping, did the Elston-Stewart algorithm start to exhibit scaling difficulties.

Outside of the capabilities of exact algorithms for linkage analysis are algorithmic approximations that employ MCMC to simulate the flow of genetic material through the pedigree. Whilst these scale to larger problems than exact algorithms, they take a long time to converge. This can be because of the long length of the Markov chain, like with Simwalk, or because of the complexity of the sampling techniques employed, like with Morgan. In a separate chapter, we detailed how two Gibbs samplers for linkage analysis operated. Both samplers use the representation of the descent graph to capture how genetic material flows from the founders down the pedigree. The locus sampler is based on the Elston-Stewart algorithm and was first implemented in the Loki software package. It samples ordered genotypes conditioned on flanking markers, which are converted into meiosis indicators. The meiosis sampler is based on the Lander-Green algorithm and was first implemented in the Morgan package. It samples the meiosis indicators at a given meiosis across all loci. Additional details like the Sobel-Lange estimator for calculating LOD scores between markers and how the Markov chain is initialised using sequential imputation were also detailed.

Across two chapters, we discussed the different hardware platforms and supporting software techniques that could be employed to implement parallel versions of the locus and meiosis samplers. We then went on to flesh out the details empirically in a series of benchmarks that informed us as to the right design decisions to make. In strictly abstract terms, there is not any difference between multicore processors and GPUs. Multicore processors contain several processor cores that can each work on indepen-

dent coarse-grained problems. Each core contains elements that can achieve fine-grain parallelism as well, e.g. hyperthreading and SIMD instructions. GPUs work on coarse-grain problems by defining them as blocks, which run on multiprocessors, and threads, which run on CUDA cores. The overlapping terminology is unfortunate, as a CUDA core, whilst general purpose, is not equivalent to the complexity or speed of a regular CPU core. From the perspective of the programmer, the main difference between the two platforms is the programming model used to develop software. On the GPU, Nvidia enforce a strict programming model that maps directly to hardware, forcing the programmer to structure the program in the correct way. This makes it clear how the system should be coded, but unfortunately requires a lot of work to port existing code. Programming on the CPU is more free-form and, whilst software informs microprocessor design, programming techniques are normally invented first with specific hardware support coming later. Our main techniques to parallelise the locus and meiosis samplers on the CPU were to identify independent coarse-grained tasks that could be run in different threads on different processor cores. In the case of the meiosis sampler, this was the likelihood of all possible founder allele assignments for a given descent graph. With the locus sampler, all markers were divided into n windows, with each window being run on a different processor core. With the graphics card, we counter-intuitively found that using the same number of threads as there were likelihood calculations in each peeling operation resulted in so much overhead that it was more efficient to use a fixed number of GPU threads for the entire peel. The complexity of the platform means that there is no way to tell what number of threads will run the fastest outside of testing them empirically. We discovered that higher numbers of threads, in excess of the work there is to do, can improve performance considerably probably due to the interplay between memory accesses and bus contention. The performance of the meiosis sampler was exceptionally poor on the GPU, prompting us to use an approximation, which proved to give inaccurate results in our later case studies.

We ran three case studies to empirically test both the accuracy and performance of

our software that we named SwiftLink. In the first case study, sensorineural deafness, we compared the results from SwiftLink with the Lander-Green algorithm-based Allegro program. The results were found to be accurate, identifying the correct regions of interest with an appropriate LOD score at varying degrees of parallelism. The second two case studies: EAST syndrome and benign chorea, were used to compare the performance of SwiftLink with Simwalk and Morgan. All versions of our program were faster and scaled well across multiple processor cores, given the limitations of Amdahl's law. Both case studies exposed how inaccurate using just the GPU was, due to the approximate windowed meiosis sampler. Fortunately, we were able to demonstrate that the GPU could still be put to use by using a hybrid approach, leveraging the advantages of both platforms by running the meiosis sampler on the CPU and the locus sampler on the GPU. This proved to be the fastest approach on the hardware we had available, but for the highly complex benign chorea pedigree suffered from a lower level of accuracy. Our experiences suggest that for pedigrees of the approximate size and complexity of the EAST syndrome case study, the hybrid approach works well. The final conclusions we came to, was that, in order to ensure the veracity of the results without the user needing to second guess whether the pedigree was too complicated or not, we should only use the GPU for performing LOD score calculations. We identified that if you want to sample at every iteration of the Markov chain and calculate LOD scores at multiple points between each consecutive pair of markers, then the computation required would actually outstrip running the Gibbs samplers for the simulation itself.

8.2 Critical Evaluation

The primary contributions of this thesis centred around three versions of our multipoint linkage analysis program, SwiftLink.

- We developed multithreaded versions of both the locus Gibbs sampler and the meiosis Gibbs sampler that, with the number of processor cores commonly available in current generation CPUs, scales well without any noticeable impact to the

accuracy of the analysis compared to single threaded implementations.

- We looked at a second parallel computing architecture: graphics processing units (GPUs) and discovered that, whilst some algorithms fit the massively parallel architecture well, others could not be implemented in a performant manner, as was the case with the meiosis sampler. It was poorly suited to the GPU because it had a particularly long sequential component proportional to the number of markers in a chromosome. Approximations attempting to alleviate this were harmful to the accuracy of results.

- Despite the failings of the GPU to perform all aspects of linkage analysis with high performance, we can leverage the fact that modern computers are increasingly heterogeneous and use both GPU and CPU in a single application. The locus sampler and LOD score code performed best on the GPU, whereas the meiosis sampler clearly only worked well on the CPU. All CUDA-based GPU applications already involve the CPU, to copy data and to orchestrate the ordering of different CUDA kernel invocations. Utilising the CPU to perform a part of the analysis it is better suited to is a natural extension.

- Finally, all our software was evaluated empirically on case studies that investigated the accuracy, run-time performance and memory requirements in scenarios of varying size. These case studies allowed us to identify where different approaches could not be applied without affecting results.

Whilst we broadly succeeded at what we set out to do, i.e. parallel linkage analysis on two different parallel computer architectures, we do not really know how much better we could do, even on the same hardware. The CPU architecture was only used in the coarsest manner possible, spreading work across multiple threads. We did not exploit

some of the finer-grain possibilities, for example, SIMD instructions could be used to accelerate the work in each thread [75]. However, it is not clear to what extent this could be exploited in our application domain nor is it clear how much time it would take to reorganise the existing code base to permit vectorisation. Similarly, our hybrid CPU / GPU application only operated in a crude manner, using either the CPU or GPU at any given time. The locus sampler and LOD scoring code runs on both platforms and at least this work could be divided evenly between the two. This is described in greater detail in the next section about future research. Even though the locus sampler and LOD score code performed well on the GPU, it could be better. For example, when performing the sampling phase of the locus sampler, many resources are idle because the sampling itself is run in a single thread. CUDA provides an interface to permit multiple *streams* of execution to be run concurrently. If we had two copies of the current descent graph, where one is always read-only and the other is write-only at any given time (similar to back-buffering in computer graphics), then the LOD scores could be calculated at the same time as the sampling process is performed in the locus sampler. In this way, we would utilise more of the hardware more of the time, though at the expense of additional complexity. One argument why it might be unwise to try to gain as much speed as possible is that to date we have only tested a single model of graphics card. We do not really know what the impact of such changes will be on other GPUs, which would have to be considered. The same argument can be made about the current version of the code. The problem is broken down in a simple manner and before each simulation short experiments are performed by the application to ensure that it is using the optimum number of threads for LOD scoring and the locus sampler individually.

One of the most obvious criticisms is that we do not provide many modes of analysis for the user, the only one being parametric linkage analysis. We had considered extending pre-existing linkage analysis software and offering the changes either as a patch-set to be applied to the source code of the other project or directly to the authors of the software. This idea was rejected in the case of Simwalk because it was written

in Fortran 77, a language that does not feature dynamic memory, making it measurably more complicated. Morgan, on the other hand, is written in C, but parts of it date back to the original Loki implementation of pedigree peeling, clearly written when memory was not as abundant as today and would have required an enormous amount of work to refine for our purposes. In addition, there are very many code paths for multiple analysis programs which we may have broken inadvertently. The final nail in the coffin was our wish to investigate GPU programming which we knew would require extensive rewriting. As it was, porting our own code, which we understood completely, was immensely difficult. Other missing features that are available in many linkage programs are: the ability to use a different genetic map for each sex to take into account the differences in recombination fractions, the ability to analyse X-linked traits, to be able to use polymorphic markers instead of SNPs and finally to calculate LOD scores at a user-defined number of positions between markers.

A criticism of all linkage analysis software is its poor ease of use. Outside of ensuring that input files are formatted correctly, SwiftLink does not prevent the user from making the wrong decisions. For example, if the user runs twice the number of threads as there are processor cores, the application will be slowed down by the needless overhead. There is not a neat solution to this outside of some system-wide oracle that understands what the computer is being used for and how many resources to allocate. As our focus is really on a single desktop PC, we default to using a single thread and otherwise make the user select the number of threads to use with a command line flag.

8.3 Application and Promotion

A majority of linkage analysis projects involving large consanguineous pedigrees with many markers either rely on the Simwalk software package or abbreviate the input data so it can be run by a Lander–Green algorithm-based program. The potential role SwiftLink can play in future genetic research will be in projects of this kind, greatly reducing analysis time. As we have shown in this thesis, SwiftLink excels at analysing

pedigrees that are similar in size to the EAST pedigree (Figure 6.1a), which is a 26-bit pedigree, but suffers with convergence problems in larger pedigrees. Certainly, any pedigree that just exceeds the capabilities of Allegro would be well suited to analysis with SwiftLink. SwiftLink has been carefully designed to slot into existing linkage analysis workflows as it accepts classical "Linkage" input files, in a similar fashion to Allegro and Merlin. SwiftLink is currently being used for projects from several research groups within University College London. As it is currently in a "beta" version, Simwalk is run at the same time and the results compared.

SwiftLink will be promoted in several ways. A manuscript is currently in the process of being written and will be submitted to a suitable journal once complete. Once the publication is in press, details will be submitted to an online directory of genetic analysis software (for example [4]).

8.4 Future Research

The biggest improvements in performance came from utilising whichever platform best suited the particular algorithm. The locus sampler and LOD scoring code ran well on the graphics card, whereas the accuracy of the meiosis sampler suffered in that situation and ran better in multiple threads on the CPU. Despite the fact this hybrid approach is already the fastest MCMC linkage analysis program, there are clear ways that it could be extended. For example, whilst both CPU and GPU platforms were used, only one was being used at any given time. When the GPU was running, the CPU was idle and vice-versa. The LOD scoring code can be run independently of the Markov chain, assuming that the underlying descent graph does not change. This could be an enormous speedup in the case where more than one LOD needs to be calculated between each pair of markers. This could even be investigated for the meiosis sampler, by performing the likelihood calculations on the GPU and copying the results to the host PC where the sampling is performed on the CPU. In a similar vein to distributing work efficiently between the CPU and GPU in the same computer, we did not address how this would be

done across multiple computers. Even with the limitations imposed by Amdahl's law, if there is available hardware, there is no reason why this cannot be used in the same analysis across a network, using, for example, MPI or MapReduce. An additional problem is that if these networked PCs had a mixture of different graphics cards with different capabilities, then how best they could be utilised would have to be dynamically decided by the running system.

There is a more abstract framework that encompasses all of these problems. Given a finite set of computing resources, where the time to complete a unit of work is defined as a function of the total run-time and the latency involved in memory copies and network signalling, what is the best way to utilise these resources to perform the task at hand? The work here is not a single linkage analysis (Amdahl's law clearly prevents us from scaling a single analysis infinitely), but it may be an entire genome-wide scan or many of them in different projects. Clearly, if we have fewer resources than jobs, then each job that gets to run should do so in the most efficient way, which is with a single thread. But as resources become idle, a currently running simulation could be signalled to expand into these resources as well. Maybe our cost function could be more complex than maximising throughput. It could involve all work being done in the most energy-efficient way possible, actively preventing hardware being utilised if the total gain in throughput is lower than a certain threshold. For example, doubling the number of processors to finish a few minutes faster is probably not worth it. In addition, such a system might take into account the earliest time that a user might actually use the results of the analysis, running on fewer processors overnight to finish at 9am when a user gets to their desk in the morning.

Another facet of the problem of how best to utilise available resources is how this is presented to the user. Whilst a few simple front-ends exist to format input files and run programs in preset pipelines, it is clear that a thorough investigation of human/computer interaction (HCI) is necessary to both manage the expectations of the user and promote

more accurate analysis. To make a case in point, there are no restrictions on running enormous numbers of SNPs in a single analysis, even though this breaks the assumption of linkage equilibrium that almost all programs implicitly make. There is a growing body of work focused on interactive design in the field of machine learning that seeks to better involve the user in the details of an algorithm, so they develop an abstract understanding of what is happening in an otherwise black box. This research would be an excellent starting point to understand how we can better present genetic analysis.

We did not investigate other MCMC schemes, such as using multiple chains, simulated tempering or parallel tempering. Our work to date on parallel samplers will be of use to any of these schemes. Multiple independent Markov chains suffer from scaling limitations due to the overhead of getting to the equilibrium distribution. Essentially, each chain must perform a suitable amount of burn-in to ensure it is sampling from the equilibrium distribution and this will dictate the length of the chain. For example, if you want to run the equivalent of 100,000 iterations of a single chain with 10% burn-in, then each chain will still need to run the same amount of burn-in followed by whatever proportion of the chain it has been assigned to run. In the case of requiring 10% of the total Markov chain iterations to be discarded as burn-in, then a 10x speedup would be the maximum possible.

Simulated and parallel tempering have an additional parameter of temperature that relaxes the model and therefore increases the acceptance rates of the Markov chain/s. Simulated tempering alters the temperature of a single chain up and down during the simulation, whereas parallel tempering has one Markov chain per temperature and exchanges states between chains. This would be one possible way to use multiple networked computers, each running their own chain at a given temperature, but each chain utilises all local processor cores. The advantages are meant to be that chains converge faster due to their being able to better explore the state space, but further experiments would be required to see if this is a large enough pay-off to invest the necessary time, especially now most genome-wide scans take only a few hours.

The final issue we have overlooked throughout this thesis is the subject of haplotype reconstruction. Haplotype reconstruction is similar to linkage analysis but whereas linkage is run as a simulation over possible descent graphs, haplotype reconstruction is an optimisation process that aims to find the most likely descent state. Simwalk uses simulated annealing to find both the initial descent graph of the Markov chain and to find the most likely descent state for haplotype reconstruction. Unfortunately, we are unaware of any approaches that use Gibbs samplers in a similar way. One possibility that we did not have time to investigate is using the temperature parameter in annealing to alter the values of the genetic model itself (i.e. the recombination fractions between markers) to find the most likely descent graph and then enumerate all possible founder alleles for that descent graph to find the most likely combination.

Bibliography

[1] G R Abecasis and J E Wigginton. “Handling marker-marker linkage disequilibrium: pedigree analysis with clustered markers”. In: *American Journal of Human Genetics* 77.5 (2005), pp. 754–767.

[2] G R Abecasis et al. “GRR: graphical representation of relationship errors”. In: *Bioinformatics* 17.8 (2001), pp. 742–743.

[3] G R Abecasis et al. “Merlin–rapid analysis of dense genetic maps using sparse gene flow trees”. In: *Nature Genetics* 30.1 (2002), pp. 97–101.

[4] *An alphabetic list of genetic analysis software*. URL: `http://www.nslij-genetics.org/soft/`.

[5] P Armitage. “Tests for linear trends in proportions and frequencies”. In: *Biometrics* 11.3 (1955), pp. 375–386.

[6] *Auto-vectorization in GCC*. URL: `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[7] M J Bamshad et al. “Exome sequencing as a tool for Mendelian disease gene discovery”. In: *Nature Reviews Genetics* 12.11 (2011), pp. 745–755.

[8] D Bockenhauer et al. “Epilepsy, ataxia, sensorineural deafness, tubulopathy, and KCNJ10 mutations”. In: *The New England Journal of Medicine* 360.19 (2009), pp. 1960–1970.

[9] C E Bonferroni. “Teoria statistica delle classi e calcolo delle probabilità”. In: *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commerciali di Firenze* 1.8 (1936), pp. 3–62.

[10] D Botstein et al. “Construction of a genetic linkage map in man using restriction fragment length polymorphisms”. In: *American Journal of Human Genetics* 32.3 (1980), pp. 314–331.

[11] A E Brockwell. “Parallel Markov chain Monte Carlo simulation by pre-fetching”. In: *Journal of Computational and Graphical Statistics* 15.1 (2006), pp. 246–261.

[12] K W Broman et al. “Comprehensive human genetic maps: Individual and sex-specific variation in recombination”. In: *American Journal of Human Genetics* 63.3 (1998), pp. 861–869.

[13] J Byrd, S Jarvis, and A Bhalerao. “On the parallelisation of MCMC-based image processing”. In: *In Proceedings of IEEE International Workshop on High Performance Computational Biology (HiCOMB)* (2010).

[14] C Cannings, E A Thompson, and H H Skolnick. “The recursive derivation of likelihoods on complex pedigrees”. In: *Advances in Applied Probability* 8.4 (1976), pp. 622–625.

[15] C Cannings, E A Thompson, and M H Skolnick. “Probability functions on complex pedigrees”. In: *Advances in Applied Probability* 10.1 (1978), pp. 26–61.

[16] V G Cheung et al. “Polymorphic variation in human meiotic recombination”. In: *American Journal of Human Genetics* 80.3 (2007), pp. 526–530.

[17] G Conant et al. “Parallel Genehunter: Implementation of a linkage analysis package for distributed-memory architectures”. In: *IEEE workshop on high performance computational biology* 16 (2002).

[18] The International HapMap Consortium. “A haplotype map of the human genome”. In: *Nature* 437.7063 (2005), pp. 1299–1320.

[19] R W Cottingham, R M Idury, and A A Schäffer. “Faster sequential genetic linkage computations”. In: *American Journal of Human Genetics* 53.1 (1993), pp. 252–263.

[20] *CPU/GPU architecture comparison*. URL: http://commons.wikimedia.org/wiki/File:Cpu-gpu.svg.

[21] *CUDA processing flow*. URL: http://commons.wikimedia.org/wiki/File:CUDA_processing_flow_(En).PNG.

[22] D J Earl and M W Deem. "Parallel tempering: theory, applications, and new perspectives". In: *Physical Chemistry Chemical Physics* 7.23 (2005), pp. 3910–3916.

[23] A P Dempster, N M Laird, and D B Rubin. "Maximum likelihood from incomplete data via the EM algorithm". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 39.1 (1977), pp. 1–38.

[24] B Devlin and N Risch. "A comparison of linkage disequilibrium measures for fine-scale mapping". In: *Genomics* 29.2 (1995), pp. 311–322.

[25] K Diefendorff et al. "AltiVec extension to PowerPC accelerates media processing". In: *IEEE Micro* 20.2 (2000), pp. 85–95.

[26] P Eastman and V Pande. "OpenMM: a hardware-independent framework for molecular simulations". In: *Computing in Science and Engineering* 12.4 (2010), pp. 34–39.

[27] R C Elston and J Stewart. "A general model for the genetic analysis of pedigree data". In: *Human Heredity* 21.6 (1971), pp. 523–542.

[28] *Erlang website*. URL: http://www.erlang.org/.

[29] M Fishelson and D Geiger. "Exact genetic linkage computations for general pedigrees". In: *Bioinformatics* 18 Suppl 1 (2002), S189–198.

[30] M Fishelson and D Geiger. "Optimizing exact genetic linkage computations". In: *Journal of Computational Biology* 11.2-3 (2004), pp. 263–275.

[31] K A Frazer et al. "A second generation human haplotype map of over 3.1 million SNPs". In: *Nature* 449.7164 (2007), pp. 851–861.

[32] D Geiger, C Meek, and Y Wexler. “A variational inference procedure allowing internal structure for overlapping clusters and deterministic constraints”. In: *Journal of Artificial Intelligence Research* 27.1 (2006).

[33] A Gelman and D Rubin. “Inference from iterative simulation using multiple sequences”. In: *Statistical Science* 7.4 (1992), pp. 457–472.

[34] S Geman and D Geman. “Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6.6 (1984), pp. 721–741.

[35] A W George et al. “Markov chain Monte Carlo methods for the calculation of likelihoods in genetic linkage studies”. In: *American Journal of Human Genetics* 69 (2001), A1337.

[36] P J Green. “Reversible jump Markov chain Monte Carlo computation and Bayesian model determination”. In: *Biometrika* 82.4 (1995), pp. 711 –732.

[37] W Gropp. “A high-performance, portable implementation of the MPI message passing interface standard”. In: *Parallel Computing* 22.6 (1996), pp. 789–828.

[38] D F Gudbjartsson et al. “Allegro, a new computer program for multipoint linkage analysis”. In: *Nature Genetics* 25.1 (2000), pp. 12–13.

[39] D F Gudbjartsson et al. “Allegro version 2”. In: *Nature Genetics* 37.10 (2005), pp. 1015–1016.

[40] J B S Haldane. “The combination of linkage values, and the calculation of distance between the loci of linked factors”. In: *Journal of Genetics* 8.4 (1919), pp. 291–297.

[41] W K Hastings. “Monte Carlo sampling methods using Markov chains and their applications”. In: *Biometrika* 57.1 (1970), pp. 97–109.

[42] S C Heath. “Markov chain Monte Carlo segregation and linkage analysis for oligogenic models”. In: *American Journal of Human Genetics* 61.3 (1997), pp. 748–760.

[43] S Hong et al. “Accelerating CUDA graph algorithms at maximum warp”. In: *Proceedings of the 16th Annual Symposium on Principles and Practice of Parallel Programming* (2011).

[44] Q Huang, S Shete, and C I Amos. “Ignoring linkage disequilibrium among tightly linked markers induces false-positive evidence of linkage for affected sib pair analysis”. In: *American Journal of Human Genetics* 75.6 (2004), pp. 1106–1112.

[45] IBM Corporation. *The Cell Architecture*. URL: `http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html`.

[46] *Id Software, Quake website*. URL: `http://www.idsoftware.com/games/quake/quake`.

[47] R M Idury and R C Elston. “A faster and more general hidden Markov model algorithm for multipoint likelihood calculations”. In: *Human Heredity* 47.4 (1997), pp. 197–202.

[48] Intel Corporation. *Pentium processors with MMX technology*. URL: `http://www.intel.com/p/en_US/embedded/hwsw/hardware/pentium-mmx`.

[49] M Irwin, N Cox, and A Kong. “Sequential imputation for multilocus linkage analysis”. In: *Proceedings of the National Academy of Sciences* 91.24 (1994), pp. 11684–11688.

[50] M Kanellos. *Moore’s law to roll on for another decade*. URL: `http://news.cnet.com/2100-1001-984051.html`.

[51] P Keleher et al. “TreadMarks: distributed shared memory on standard workstations and operating systems”. In: *In proceedings of the 1994 winter USENIX conference* (1994), pp. 115–131.

[52] A P Klein et al. “Investigation of altering single-nucleotide polymorphism density on the power to detect trait loci and frequency of false positive in non-parametric linkage analyses of qualitative traits”. In: *BMC Genetics* 6.Suppl 1 (2005), S20–S20.

[53] R J Klein et al. “Complement factor H polymorphism in age-related macular degeneration”. In: *Science* 308.5720 (2005), pp. 385–389.

[54] A Kong. “Analysis of pedigree data using methods combining peeling and Gibbs sampling”. In: *Computer Science and Statistics: Proceedings of the 23rd Symposium on the Interface* (1991), pp. 379–385.

[55] A Kong et al. “A high-resolution recombination map of the human genome”. In: *Nature Genetics* 31.3 (2002), pp. 241–247.

[56] A Kong et al. “Fine-scale recombination rate differences between sexes, populations and individuals”. In: *Nature* 467.7319 (2010), pp. 1099–1103.

[57] L Kruglyak, M J Daly, and E S Lander. “Rapid multipoint linkage analysis of recessive traits in nuclear families, including homozygosity mapping”. In: *American Journal of Human Genetics* 56.2 (1995), pp. 519–527.

[58] L Kruglyak and E S Lander. “Complete multipoint sib-pair analysis of qualitative and quantitative traits”. In: *American Journal of Human Genetics* 57.2 (1995), pp. 439–454.

[59] L Kruglyak and E S Lander. “Faster multipoint linkage analysis using Fourier transforms”. In: *Journal of Computational Biology* 5.1 (1998), pp. 1–7.

[60] L Kruglyak et al. “Parametric and nonparametric linkage analysis: a unified multipoint approach”. In: *American Journal of Human Genetics* 58.6 (1996), pp. 1347–1363.

[61] E Lander and L Kruglyak. “Genetic dissection of complex traits: guidelines for interpreting and reporting linkage results”. In: *Nature Genetics* 11.3 (1995), pp. 241–247.

[62] E S Lander and P Green. “Construction of multilocus genetic linkage maps in humans”. In: *Proceedings of the National Academy of Sciences* 84.8 (1987), pp. 2363–2367.

[63] E S Lander and N J Schork. “Genetic dissection of complex traits”. In: *Science* 265.5181 (1994), pp. 2037–2048.

[64] E S Lander et al. “Initial sequencing and analysis of the human genome”. In: *Nature* 409.6822 (2001), pp. 860–921.

[65] K Lange and M Boehnke. “Extensions to pedigree analysis. V. Optimal calculation of Mendelian likelihoods”. In: *Human Heredity* 33.5 (1983), pp. 291–301.

[66] K Lange and S Matthysse. “Simulation of pedigree genotypes by random walks”. In: *American Journal of Human Genetics* 45.6 (1989), pp. 959–970.

[67] K Lange and E Sobel. “A random walk method for computing genetic location scores.” In: *American Journal of Human Genetics* 49.6 (1991), pp. 1320–1334.

[68] G M Lathrop et al. “Multilocus linkage analysis in humans: detection of linkage and estimation of recombination”. In: *American Journal of Human Genetics* 37.3 (1985), pp. 482–498.

[69] G M Lathrop et al. “Strategies for multilocus linkage analysis in humans”. In: *Proceedings of the National Academy of Sciences* 81.11 (1984), pp. 3443–3446.

[70] S L Lauritzen and N A Sheehan. “Graphical models for genetic analysis”. In: *Statistical Science* 18.4 (2003), pp. 489–514.

[71] R Leadbetter. *The legacy of id software*. URL: `http://www.gamesindustry.biz/articles/digitalfoundry-the-legacy-of-id-software`.

[72] A Lee et al. “On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods”. In: *Journal of Computational and Graphical Statistics* 19.4 (2009).

[73] E A Lee. “The Problem with Threads”. In: *IEEE Computer* 39.5 (2006), pp. 33–42.

[74] S H Lee, J H J Van der Werf, and B Tier. “Combining the meiosis Gibbs sampler with the random walk approach for linkage and association studies with a general complex pedigree and multimarker loci”. In: *Genetics* 171.4 (2005), pp. 2063–2072.

[75] V Lee et al. “Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News, ISCA ’10* 38.3 (2010), pp. 451–460.

[76] S Lin and T P Speed. “Incorporating crossover interference into pedigree analysis using the χ^2 model”. In: *Human Heredity* 46.6 (1996), pp. 315–322.

[77] M J Litzkow, M Livny, and M W Mutka. “Condor: a hunter of idle workstations”. In: *Distributed Computing Systems*. 1988, pp. 104–111.

[78] C Liu et al. “SOAP3: ultra-fast GPU-based parallel alignment tool for short reads”. In: *Bioinformatics* 28.6 (2012), pp. 878–879.

[79] *Major events in meiosis*. URL: `http://commons.wikimedia.org/wiki/File:MajorEventsInMeiosis_variant.svg?uselang=en-gb`.

[80] K Markianos, M J Daly, and L Kruglyak. “Efficient multipoint linkage analysis through reduction of inheritance space”. In: *American Journal of Human Genetics* 68.4 (2001), pp. 963–977.

[81] M Matsumoto and T Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (1998), pp. 3–30.

[82] R Merritt. *CPU designers debate multi-core future*. URL: `http://eetimes.com/electronics-news/4076123/CPU-designers-debate-multi-core-future`.

[83] N Metropolis. “The beginning of the Monte Carlo method”. In: *Los Alamos Science* 15 (1987), pp. 125–130.

[84] N Metropolis et al. “Equation of state calculations by fast computing machines”. In: *The Journal of Chemical Physics* 21.6 (1953), p. 1087.

[85] *Microsoft Direct X website*. URL: `http://www.microsoft.com/games/en-gb/aboutGFW/pages/directx.aspx`.

[86] *Microsoft DirectCompute website*. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx.

[87] G E Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (Apr. 1965), p. 114.

[88] T H Morgan. "Sex limited inheritance in drosophila". In: *Science* 32.812 (1910), pp. 120–122.

[89] *Morgan 3 Tutorial*. URL: http://www.stat.washington.edu/thompson/Genepi/MORGAN/morgan303-tut-html/morgan-tut_8.html\#SEC57.

[90] N E Morton. "Sequential tests for the detection of linkage". In: *American Journal of Human Genetics* 7.3 (1955), pp. 277–318.

[91] N Mukhopadhyay et al. "Mega2: data-handling for facilitating genetic linkage and association analyses". In: *Bioinformatics* 21.10 (2005), pp. 2556–2557.

[92] *NHGRI Karyogram*. URL: http://commons.wikimedia.org/wiki/File:NHGRI_human_male_karyotype.png.

[93] *Nvidia CUDA C programming guide, version 4.0*. URL: http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[94] *Nvidia CUDA website*. URL: http://www.nvidia.com/object/cuda_home_new.html.

[95] J R O'Connell and D E Weeks. "PedCheck: a program for identification of genotype incompatibilities in linkage analysis". In: *American Journal of Human Genetics* 63.1 (1998), pp. 259–266.

[96] J R O'Connell and D E Weeks. "The VITESSE algorithm for rapid exact multilocus linkage analysis via genotype set-recoding and fuzzy inheritance". In: *Nature Genetics* 11.4 (1995), pp. 402–408.

[97] *OpenCL website*. URL: http://www.khronos.org/opencl/.

[98] *OpenGL website*. URL: http://www.opengl.org/.

[99] J Ott. “A computer program for linkage analysis of general human pedigrees”. In: *American Journal of Human Genetics* 28.5 (1976), pp. 528–529.

[100] J Ott. “Estimation of the recombination fraction in human pedigrees: efficient computation of the likelihood for human linkage studies.” In: *American Journal of Human Genetics* 26.5 (1974), pp. 588–597.

[101] J Ott. “Maximum likelihood estimation by counting methods under polygenic and mixed models in human pedigrees.” In: *American Journal of Human Genetics* 31.2 (1979), pp. 161–175.

[102] J C Pérez-Póveda, L G Palacio, and M Arcos-Burgos. “Description of an endogamous, multigenerational and extensive family with benign hereditary chorea from the Paisa community”. In: *Revista de Neurologia* 41.2 (2005), pp. 95–98.

[103] M Plummer et al. “CODA: convergence diagnosis and output analysis for MCMC”. In: *R News* 6.1 (2006), pp. 7–11.

[104] L R Rabiner. “A tutorial on hidden Markov models and selected applications in speech recognition”. In: *Proceedings of the IEEE* 77 (1989), pp. 257–286.

[105] F Rüschendorf and P Nürnberg. “ALOHOMORA: a tool for linkage analysis using 10K SNP array data”. In: *Bioinformatics* 21.9 (2005), pp. 2123–2125.

[106] F Sanger, S Nicklen, and A R Coulson. “DNA sequencing with chain-terminating inhibitors”. In: *Proceedings of the National Academy of Sciences* 74.12 (1977), pp. 5463–5467.

[107] *Scala website*. URL: `http://www.scala-lang.org/`.

[108] A A Schäffer et al. “Avoiding recomputation in linkage analysis”. In: *Human Heredity* 44.4 (1994), pp. 225–237.

[109] N Sheehan and A Thomas. “On the irreducibility of a Markov chain defined on a space of genotype configurations by a sampling scheme”. In: *Biometrics* 49.1 (1993), pp. 163–175.

[110] M Silberstein et al. “Online system for faster multipoint linkage analysis via parallel execution on thousands of personal computers”. In: *American Journal of Human Genetics* 78.6 (2006), pp. 922–935.

[111] E Sobel and K Lange. “Descent graphs in pedigree analysis: applications to haplotyping, location scores, and marker-sharing statistics”. In: *American Journal of Human Genetics* 58.6 (1996), pp. 1323–1337.

[112] E Sobel and K Lange. “Metropolis sampling in pedigree analysis”. In: *Statistical Methods in Medical Research* 2.3 (1993), pp. 263–282.

[113] E Sobel, H Sengul, and D E Weeks. “Multipoint estimation of identity-by-descent probabilities at arbitrary positions among marker loci on general pedigrees”. In: *Human Heredity* 52.3 (2001), pp. 121–131.

[114] H C Stanescu et al. “Risk HLA-DQA1 and PLA2R1 alleles in idiopathic membranous nephropathy”. In: *The New England Journal of Medicine* 364.7 (2011), pp. 616–626.

[115] A H Sturtevant. “The linear arrangement of six sex-linked factors in drosophila, as shown by their mode of association”. In: *Journal of Experimental Zoology* 14.1 (1913), pp. 43–59.

[116] M A Suchard and A Rambaut. “Many-core algorithms for statistical phylogenetics”. In: *Bioinformatics* 25.11 (2009), pp. 1370–1376.

[117] H Sutter. “The free lunch is over”. In: *Dr Dobb’s Journal* 30.3 (2005).

[118] R H Swendsen and J Wang. “Replica Monte Carlo simulation of spin-glasses”. In: *Physics Review Letters* 57 (21 1986), pp. 2607–2609.

[119] A Thomas. “Optimal computation of probability functions for pedigree analysis”. In: *Mathematical Medicine and Biology* 3.3 (1986), pp. 167–178.

[120] E A Thompson. *Morgan*. URL: `http://www.stat.washington.edu/thompson/Genepi/MORGAN/Morgan.shtml`.

[121] E A Thompson and S C Heath. “Estimation of conditional multilocus gene identity among relatives”. In: *Lecture Notes-Monograph Series* 33 (1999), pp. 95–113.

[122] L Tong and E A Thompson. “Multilocus LOD scores in large pedigrees: combination of exact and approximate calculations”. In: *Human Heredity* 65.3 (2008), pp. 142–153.

[123] J C Venter et al. “The sequence of the human genome”. In: *Science* 291.5507 (2001), pp. 1304–1351.

[124] A Viterbi. “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”. In: *IEEE Transactions on Information Theory* 13.2 (1967), pp. 260–269.

[125] P D Vouzis and N V Sahinidis. “GPU-BLAST: using graphics processors to accelerate protein sequence alignment”. In: *Bioinformatics* 27.2 (2011), pp. 182–188.

[126] E M Wijsman, J H Rothstein, and E A Thompson. “Multipoint linkage analysis with many multiallelic or dense diallelic markers: Markov chain-Monte Carlo provides practical approaches for genome scans on general pedigrees”. In: *American Journal of Human Genetics* 79.5 (2006), pp. 846–858.

[127] R Winterhalter. *Why Quake changed games forever*. URL: `http://www.1up.com/features/why-quake-changed-games-forever`.

A Appendix

A.1 Manual

SwiftLink is a program for multipoint linkage analysis on large pedigrees using Gibbs sampling written in C++ for Linux and distributed under the GNU Public License version 3 (GPL3) with no warranty. For more details about the licence see `http://www.gnu.org/licenses/gpl-3.0.html`.

A.1.1 Installation

SwiftLink requires that the following software and libraries be installed: gcc (at least version 4.2 to use OpenMP, tested with version 4.4.3), git, make, GNU scientific library (version 1.13 tested) and CUDA 4.0 development kit. SwiftLink has only been tested on Nvidia graphics cards that had compute capacity 2.0 or better. The additional tools require python (version 2.6.5 tested) and gnuplot (version 4.2 tested). SwiftLink has only been tested on Ubuntu GNU/Linux 64-bit using the 2.6.32-39-server kernel.

The current version of SwiftLink is available from the author's GitHub repository at: `https://github.com/ajm/swiftlink`. A copy of the complete repository can be downloaded using Github's web interface or from the command line with:

```
git clone git://github.com/ajm/swiftlink.git
```

All versions are compiled with:

```
make
```

It is possible (almost approaching certain) that you will have to edit the Makefile, just a little, to indicate the locations of different libraries. In addition, all binaries must be manually copied to a directory in your PATH. The build system will be improved at a later date.

A.1.2 Options

SwiftLink contains options to tell it the locations of input files and how to run the simulation. There are two binaries: **swift32** and **swift64**, both have the same options, but we recommend that swift32 be used with a graphics card and swift64 used on the CPU.

SwiftLink has the following options:

-p pedfile --pedigree=pedfile

Indicates the location of the pedigree file. See Section A.1.4 for more details. This option is mandatory.

-m mapfile --map=mapfile

Indicates the location of the map file. See Section A.1.4 for more details. This option is mandatory.

-d datfile --dat=datfile

Indicates the location of the linkage-style data file. See Section A.1.4 for more details. This option is mandatory.

-o outfile --output=outfile

Sets the name of the output file for the linkage analysis results. The default output file is called “linkage.out” in the current working directory.

-i NUM --iterations=NUM

Sets the number of iterations that the Markov chain will be run for. Note that this number does not include any burn-in, which is set separately. The default number of iteration is 90,000.

-b NUM --burnin=NUM

Sets the number of burn-in iterations that are run before the simulation starts. The default number of burn-in iterations is 10,000.

-s NUM --sequentialimputation=NUM

Sets the number of iterations of sequential imputation that are performed to find a good starting state for the Markov chain. The default number of sequential imputation iterations is 1,000.

-x NUM --scoringperiod=NUM

Sets the scoring period, which states how often to sample from the Markov chain to calculate LOD scores. For example, setting the scoring period to 10 will score every 10^{th} iteration. The default scoring period is 1.

-l FLOAT --lsamplerprobability=FLOAT

Sets the probability that the locus sampler is selected at each iteration of the Markov chain. The default locus sampler probability is 0.5.

-c NUM --cores=NUM

Sets the number of threads that will be spawned in parallel. The default number of threads is 1. This should not be set higher than the number of processor cores, as it will just result in wasted overhead.

-g --gpu

If set, then SwiftLink will use the primary GPU.

-v --verbose

Increasing verbosity provides additional information as the program is running, such as how it has interpretted the input files and various warnings.

-h --help

Displays information about options for SwiftLink.

A.1.3 Examples

A basic run of SwiftLink using 100 iterations of sequential imputation and 30,000 iterations of MCMC, of which 10% are burn-in, and run in a single thread:

```
swift64 -p pedin.01 -m map.01 -d datain.01
        -s 100 -i 27000 -b 3000
```

The same as before, but now running in four CPU threads and specifying an results file called "results.01":

```
swift64 -p pedin.01 -m map.01 -d datain.01
        -s 100 -i 27000 -b 3000 -c 4 -o results.01
```

Using the 32-bit version of SwiftLink with the GPU and four CPU threads on the same analysis:

```
swift32 -p pedin.01 -m map.01 -d datain.01
        -s 100 -i 27000 -b 3000 -c 4 -g
```

A.1.4 File Formats

All input files are currently identical to the Linkage program. Whilst we could have designed some more intuitive file formats, many modern programs, e.g. Allegro and several others, do the same thing. The main advantage of supporting the Linkage format of input files, is that user can already use existing tools to format their data, e.g. Alohomora. Merlin currently supports both the original Linkage format and its own custom format. In the future, it is likely that we will either support the Merlin format, or try to design something better.

All files can include comments by using the hash (#) symbol. All characters after the hash are ignored by SwiftLink.

Pedigree File

The pedigree file contains information about the individuals that will be analysed. Each line corresponds to an individual. An individual is described using six fields followed by two fields per genotype. The first six fields are identifiers of: pedigree, individuals id, fathers id, mothers id, gender and affection status.

All columns must be positive numeric values. The *pedigree* field is an identifier to state which pedigree the individual belongs to. A pedigree file can contain an arbitrary number of pedigrees, they will each be examined in turn. The *id* field uniquely identifiers the individual, this can be any positive integer with the exception of zero, as zero is reserved to indicate individuals not found in the pedigree, like the parents of founders. The next two columns are the identifiers of the individuals *father* and *mother*, respectively. If the individual is a founder, then their parents are both 0. Individuals can be put in any order in a pedigree file, so individuals can forward reference parents that have not been defined so far in the file. The last two columns are *sex* and *affection status*. Sex has three possible values: 0 for unknown, 1 for male and 2 for female. Affections status has three possible values as well: 0 for unknown affection, 1 for unaffected and 2 for affected.

As an example, take the pedigree in Figure A.1. There are six individuals in the pedigree, so each will get its own line. There is only a single pedigree so we have made everyone belong to pedigree 1 by setting that as the value in the pedigree column. Each individual is numbered 1–6 in column 2 (it does not matter that we have pedigree 1 and individual 1, in fact, if there was a second pedigree, we can safely reuse all the identifiers from the first pedigree). In columns 3 and 4, the founders parents are both defined as 0 and all non-founders are the offspring of other individuals in the pedigree. Column 5, sex, shows that individuals 1,3,5 and 6 are male and the rest are female and the final column, shows that individuals 4 and 6 are unaffected, 2,3 and 5 are affected and 1 is of unknown affection.

After the six fields describing the individuals, each line must contain a further $2n$ columns, where n is the number of markers from genotyping. The markers are assumed

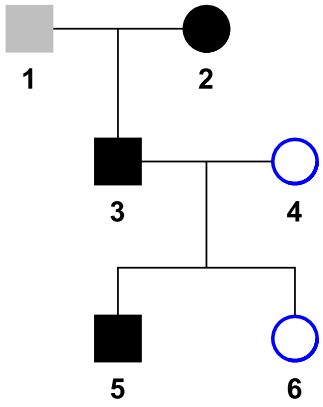


Figure A.1: Simple three generation pedigree.

pedigree	id	father	mother	sex	affection
1	1	0	0	1	0
1	2	0	0	2	2
1	3	1	2	1	2
1	4	0	0	2	1
1	5	3	4	1	2
1	6	3	4	1	1

to be in the same order as those found in the map and data files. The value of each allele can be encoded as 0 for untyped and 1 or 2 for each of the alleles of that marker. At the present time we only support biallelic SNPs.

Map File

The map file contains 5 columns for the chromosome, genetic position, marker name, physical position and an index. Genetic position is measured in centiMorgans and physical position is in basepairs.

chromosome	genetic position	marker name	physical position	index
1	3.456	rs1	3088998	1
1	5.697	rs2	4215064	2
1	8.056	rs3	5034491	3

Data File

The data file from the original Linkage program provided the control options for each analysis, stating which program should be run and how. We actually ignore most of the information contained in this file and rely on command line flags to tell SwiftLink what to do, but some of it is very important, for example, the map file did not contain any information about the allele frequencies of each marker.

There are three sections in the data file: loci information, marker descriptions and recombination information. We will only describe the details of the fields actually used by SwiftLink, for further details about the specification please see Section 2.6 of the Linkage manual: `http://linkage.rockefeller.edu/soft/linkage/sec2.6.html`

Loci information is the first three lines of the data line. The first line contains four fields: number of loci, risk locus, sex linked and program number. The only two that are important for SwiftLink is the number of loci, that should be set to the number of SNPs $+1$ (for the disease trait) and sex linked, that must be set to 0 as we have only implemented autosomal analysis so far. The second line contains another four fields: mutant locus, male mutation rate, female mutation rate and linkage disequilibrium. All of these must be set to 0. The third line is the ordering of markers in the analysis, SwiftLink assumes that the order of markers in the file is already correct. An example of the loci information for the map file example from the last section is as follows:

```
4 0 0 5
0 0.0 0.0 0
1 2 3
```

The marker descriptions contain the details of the allele frequencies of the disease trait and SNPs. Each marker has at least two lines. The first line contains two fields: the marker type and the number of alleles. The only marker types we support are affection status (1) and numbered alleles (3). The second line must have the number of fields as there were alleles stated in the first line. For both affection status and numbered alleles, this will always be two, as SwiftLink only supports SNPs. Affection status contains two

extra lines that markers do not, the third line must be a 1 (this is the number of liability classes). The fourth line contains a description of that liability class, which is basically a probability for each combination of alleles, the number of which were stated in the first line. In the case of the affection status, the liability class is actually the penetrance function. For a fully penetrant dominant trait you would use ”0 1 1” and for a fully penetrant recessive trait ”0 0 1”.

An example of a fully penetrant recessive trait with a minor allele frequency of 0.0001 is as follows:

```
1 2 # TRAIT
0.9999 0.0001
1
0.0 0.0 1.0
```

The three markers in the map file we described previously, can be described:

```
3 2 # rs1
0.875 0.125
3 2 # rs2
0.225 0.775
3 2 # rs3
0.01 0.99
```

The final section is for recombination information. It has three lines, the first line has two fields: sex difference and interference, which must both be set to 0. The second line states the recombination fractions between each pair of markers as it appeared in the file. The first recombination fraction is for the trait, SwiftLink ignores it. All other recombination fractions are for each pair of consecutive markers, therefore there should be $n - 1$ recombination fractions for n markers. The third line states values for the difference in recombination fractions for males and females, but it is unused. The recombination information for our running example is given as:

```
0 0
0.1 0.021913 0.023040
1 2.0 1.0
```

A.2 MCMC Diagnostics

In section 6.3.2 we used both the autocorrelation statistic and the Gelman–Rubin diagnostic to aid us in assessing the convergence of different Markov chains.

A.2.1 Autocorrelation Statistic

To assess the degree of correlation between successive approximations ξ, we use the following autocorrelation function to calculate the k^{th} order (lag) correlation:

$$\rho_k = \frac{\sum_{i=1}^{n-k}(\xi_i - \bar{\xi})(\xi_{i+k} - \bar{\xi})}{\sum_{i=1}^{n}(\xi_i - \bar{\xi})^2}$$

Autocorrelation is essentially a correlation coefficient between two values of the same variable at points i and $i + k$ in the series. An autocorrelation plot is used to display autocorrelations at varying time lags.

A.2.2 Gelman–Rubin Diagnostic

The Gelman–Rubin diagnostic is calculated over multiple chains that began from different starting states. Each chain is run for $2n$ iterations and the first n iterations are discarded. The goal is to calculate the in-chain variance and the between-chain variance, if the chains converge then the variance should be low. This is assessed by calculating the potential scale reduction factor.

We calculate the in-chain variance, W, as:

$$W = \frac{1}{m}\sum_{j=1}^{m} s_j^2$$

here W is the mean of the variances of m chains and the variance of the j^{th} chain s is

$$s_j^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\theta_{ij} - \bar{\theta_j})^2$$

The between-chain variance is defined as

$$B = \frac{n}{m-1}\sum_{j=1}^{m}(\bar{\theta_j} - \bar{\bar{\theta}})^2$$

where

$$\bar{\bar{\theta}} = \frac{1}{m}\sum_{j=1}^{m}\bar{\theta_j}$$

The variance of the equilibrium distribution is estimated as

$$\widehat{Var}(\theta) = (1 - \frac{1}{n})W + \frac{1}{n}B$$

The potential scale reduction factor is

$$\hat{R} = \sqrt{\frac{\widehat{Var}(\theta)}{W}}$$

where $\hat{R}$ has a high value, we use greater than 1.05, it is suggestive that the chain may not have converged and therefore needs to be run for longer.